

AQUA Guardian AI-power multi-robot coordination defence for maritime security

By

Sharique Hassan Malik

Enrollment No. 01-133222-071

Hamna Zahra

Enrollment No. 01-133222-024

Supervised By

Dr. Nadia Sultan

Co Supervised By

Dr. Muhammad Irfan



Session 2022-26

A Report is submitted to the Department of Electrical Engineering,
Bahria University, Islamabad.

In partial fulfillment of requirement for the degree of BS(EE).

Certificate

We accept the work contained in this report as a confirmation to the required standard for the partial fulfillment of the degree of BS(EE).

Head of Department

Supervisor

Internal Examiner

External Examiner

Dedication

We are dedicating this to our beloved parents who have offered unending support and sacrifices to us and whose encouragement has been the pillar of our academic life. This achievement has been made possible by their faith in our capabilities, which has been a source of strength and inspiration.

We thank our esteemed supervisors, Dr. Nadia Sultan and Dr. Muhammad Irfan, who provided us with invaluable guidance, insightful feedback and support along with this project.

Special and warm recognition is given to our supervisor, Dr. Nadia Sultan, whose outstanding dedication, continuous motivation and relentless effort were crucial in the development of this work. Her encouragement was much more than we expected and we are so grateful that she was with us through it all and giving us good advice.

Acknowledgments

To begin with, it is our utmost gratitude to Almighty Allah, whose inmost mercy, direction and blessings made us come out of this project successfully. This would not have been achievable without His aid.

We wish to show our deepest appreciation to our supervisors, Dr. Nadia Sultan and Dr. Muhammad Irfan who have been supportive, given us expert advice and encouraged us during this project. Their input and suggestions played a significant role in the development of this work.

We are grateful to the Department of Electrical Engineering and Center of Excellence in AI, Bahria University, who helped us with the required resources. We also want to thank NESCOM who funded this project and offered technical assistance.

We would like to recognize the help of our colleagues and lab personnel that helped to complete this project successfully.

Abstract

Coordinating a fleet of autonomous robots to allocate tasks and plan paths intelligently remains one of the hardest open problems in multi-agent AI. Existing approaches either handle task allocation or motion planning in isolation and rule-based systems break down the moment the environment changes faster than the rules anticipate. AQUA Guardian tackles this by combining both layers into a single end-to-end framework.

Task allocation is handled by the HybridRL agent, which integrates MAPPO, QMIX, MADDPG and DQN. Each algorithm contributes to the final dispatching decision and a performance-weighted selection mechanism aggregates their outputs based on cumulative task outcomes. Robots are aware of task dependencies, so no robot is sent to a job before its prerequisites are complete. Path planning operates through the HybridDiffusion model, built from TDM, DiffusionPolicy, Diffuser and SDE running in parallel. Each planner generates candidate trajectories that are scored and fused into a single smooth, collision-free path.

The framework is simulated in Gazebo Harmonic and ROS 2 Jazzy running on Ubuntu 24.04, and evaluated across three scenarios of increasing complexity. These span warehouse logistics, search-and-rescue and open-water maritime patrol. Task completion rate and time to finish capture operational performance while path quality and energy use reflect trajectory efficiency. The system is built to support both offline and online training.

Contents

1	Introduction	1
1.1	Project Background and Overview	2
1.2	Problem Description	5
1.3	Problem Statement	7
1.4	Project Objectives	8
1.5	Project Scope	10
2	Literature Review	13
2.1	Multi-Agent Reinforcement Learning	14
2.2	Diffusion Models for Robot Motion Planning	18
2.3	Multi-Robot Systems and Coordination Architectures	22
2.4	Task Allocation Strategies	24
2.5	Motion and Path Planning for Multi-Robot Systems	25
2.6	Foundation Models and Theoretical Underpinnings	27
2.7	Research Gaps and Positioning of AQUA Guardian	28
3	Requirement Specifications	30
3.1	Existing System	31
3.2	Proposed System	32
3.3	Requirement Specifications	35
3.3.1	Functional Requirements	35
3.3.2	Non-Functional Requirements	36
3.4	Use Cases	37
3.4.1	Warehouse Logistics Coordination	37
3.4.2	Search-and-Rescue Response	39

3.4.3	Maritime Security Patrol and Response	40
4	System Design	43
4.1	System Architecture	44
4.2	Design Constraints	46
4.3	Design Methodology	48
4.4	High Level Design	49
4.5	Low Level Design	52
4.5.1	Attention Encoder for the RL Module	52
4.5.2	Score Network for the Diffusion Module	53
4.5.3	Cost-Guided DDIM Sampling	54
4.5.4	Action Masking	55
4.5.5	Reward Shaping	55
4.6	Database Design	56
4.7	GUI Design	57
4.8	External Interfaces	58
5	System Implementation	60
5.1	System Architecture	61
5.2	Tools and Technology Used	62
5.3	Development Environment/Languages Used	64
5.4	Processing Logic/Algorithms	65
5.4.1	RL Training Loop	65
5.4.2	Diffusion Training Loop	66
5.4.3	HybridRL Bandit Selection	67
5.4.4	Cost-Guided DDIM Inference	67
5.4.5	Job Orchestration	68
5.5	Application Access Security	68
5.6	Database Security	69
6	System Testing and Evaluation	71
6.1	Testing Strategy	72
6.2	Evaluation Metrics	73

6.3	Experimental Protocol	74
6.4	Results	75
6.4.1	Cross-Algorithm Comparison	75
6.4.2	Hybrid versus Base Comparison	83
6.4.3	Convergence Behaviour	86
6.4.4	HybridRL Bandit Behaviour	92
6.4.5	Scalability	93
6.4.6	Summary Table	94
6.5	Limitations of the Evaluation	95
7	Conclusion	99
7.1	Summary of Contributions	100
7.2	Empirical Findings and Their Implications	101
7.3	Limitations	102
7.4	Future Work	103
7.5	Closing Remarks	105
	References	107
A	User Manual	119
B	Model Training Code	134
C	Simulation Code	672

List of Figures

1.1	Multi-robot coordination problem overview	8
1.2	AQUA Guardian project objectives summary	10
3.1	Key robot parameters across simulation scenarios	33

3.2	Warehouse scenario layout	38
3.3	Search-and-rescue scenario map	40
3.4	Maritime patrol area layout	42
4.1	AQUA Guardian four-layer system architecture	45
4.2	Single environment step data flow	50
4.3	Four-stage training pipeline with barriers	51
4.4	Attention encoder task dependency graph	53
4.5	B-spline vs. dense waypoint parameterisation	54
4.6	DDIM denoising progression (20 steps)	55
6.1	RL task allocation frequency heatmap	77
6.2	Warehouse task spawn-location heatmap	78
6.3	SAR task spawn-location heatmap	79
6.4	Maritime task spawn-location heatmap	80
6.5	Warehouse episode reward distributions	81
6.6	SAR episode reward distributions	81
6.7	Maritime episode reward distributions	82
6.8	Combined RL×diffusion performance heatmap	83
6.9	RL algorithm computational complexity analysis	85
6.10	HybridDiffusion vs. base diffusion best loss	86
6.11	RL training learning curves	87
6.12	Diffusion training-loss curves	89
6.13	HybridDiffusion trajectory output	90
6.14	Inter-robot collision avoidance via cost-guided DDIM	91
6.15	Effect of task conditioning on trajectory distribution	91
6.16	HybridRL UCB bandit selection profile	93
6.17	Scalability: completion rate vs. fleet size	94

List of Tables

6.1	RL training results across all scenarios	76
6.2	HybridRL vs. base RL completion rate comparison	84
6.3	HybridDiffusion vs. base diffusion loss comparison	86
6.4	Diffusion training results across all scenarios	88
6.5	Warehouse scenario: all algorithm results	95
6.6	SAR scenario: all algorithm results	95
6.7	Maritime scenario: all algorithm results	96
6.8	Complete raw results: all 30 trained configurations	96
6.9	Consolidated headline metrics: all algorithms and scenarios	97

Chapter 1

Introduction

1.1 Project Background and Overview

The maritime domain has always posed singular challenges to human operators. Vast and largely featureless to the naked eye, it is in practice anything but uniform. Submarine topography varies across depths that no single sensor covers. Surface currents shift on hourly timescales that resist accurate forward prediction. Weather windows open and close on an unpredictable calendar. The operating area also contains both friendly and hostile actors whose intent is rarely declared. These conditions make the maritime environment one of the most operationally demanding on the planet. Navies and coast guards have historically relied on crewed vessels, fixed shore-based radar networks and periodic aerial reconnaissance to maintain situational awareness across contested or high-risk waters. The historical methods are expensive to sustain. They are also slow to redeploy and are increasingly outmatched by the growing pace of unmanned and semi-autonomous threats [1]. Recent deep learning approaches tailored to autonomous surface vehicles have begun to bridge the response-time gap, offering real-time perception and guidance capabilities that were previously unachievable without human operators [2].

Unmanned surface vehicles, unmanned aerial vehicles and unmanned underwater vehicles have each matured to the point where individual deployment is no longer remarkable. The swarm paradigm has attracted substantial research interest, with multiple recent surveys documenting rapid capability advances alongside persistent challenges in scalability and robustness [3–5]. What remains genuinely difficult is making heterogeneous fleets of such platforms act with genuine coordination. Coordination here means three distinct things at once. The fleet must share intelligence across agents with limited communication bandwidth. It must redistribute tasks

under uncertainty about both environmental state and the operational status of peer agents. It must generate safe trajectories in real time without constant human intervention. That triple requirement is the motivating concern of the project described in this report.

AQUA Guardian is an academic and applied research project that develops a unified computational framework for intelligent multi-robot coordination oriented toward maritime defence operations. The project draws on two complementary streams of recent machine learning research. One stream is multi-agent reinforcement learning, which provides the decision-making layer needed to allocate tasks among robots with differing capabilities and positional constraints [6, 7]. The other stream is generative diffusion modelling, which supplies a principled mechanism for producing smooth, physically feasible and collision-free trajectories even in environments dense with obstacles and moving agents [8, 9]. Neither methodology alone is sufficient. Their structured integration constitutes the project’s central technical contribution.

The choice of reinforcement learning as the backbone of the task allocation module reflects a well-established consensus in the robotics community [10, 11]. Policy-gradient and value-based methods have demonstrated the capacity to learn cooperative strategies in partially observable, multi-agent settings without requiring exhaustive hand-crafting of coordination rules. The choice of diffusion models for motion planning is more recent and arguably more provocative. Diffusion-based planners sidestep two persistent weaknesses of traditional optimisation-based planners. Local minima trap classical methods in narrow corridors where the gradient of the cost function points in misleading directions. Unimodal representations also fail to capture the full distribution of viable trajectories when multiple paths exist around an obstacle field [12]. By conditioning a generative trajectory

distribution on task assignments produced upstream by the MARL module, AQUA Guardian aims to exploit the strengths of both methods while partially offsetting their individual limitations.

To stress-test generality rather than optimise for a single scenario, the framework is evaluated across three qualitatively distinct environments. Warehouse logistics operates in a structured grid with dense static obstacles and predictable demand profiles. Search-and-rescue (SAR) operates in a procedurally generated disaster-area map with unknown survivor locations. Open-water maritime security patrol operates in a large, open environment with moving contacts and continuous-time surface vehicle dynamics. Each scenario imposes a different spatial structure, a different reward landscape and a different failure mode. Performance across all three provides stronger evidence of architectural robustness than a single-domain evaluation alone [13].

Within each scenario the framework trains five reinforcement learning algorithms and five diffusion planning algorithms. The four base RL algorithms are MAPPO [14], QMIX [15], MADDPG [16] and DQN [17]. A composite HybridRL configuration draws on properties of all four through a runtime upper-confidence-bound bandit [18] that warm-starts from the four base checkpoints. The four base diffusion planners are a Trajectory Diffusion Model derived from DDPM [19], DiffusionPolicy [20], Diffuser [21] and a Variance-Exploding stochastic differential equation variant [22]. A composite HybridDiffusion model runs all four base planners in parallel and selects the highest-scoring trajectory through a learned cost function. The diffusion models are trained on top of the HybridRL agent rather than against every cross-product of RL and diffusion algorithms, since training the full seventy-five-pair grid would multiply compute by a factor of five without adding architectural information beyond what the fo-

cused thirty-configuration training already produces. The resulting configuration database supports empirical comparison of base algorithms against their hybrid extensions and across the three scenario types.

1.2 Problem Description

Contemporary maritime security infrastructure rests on a set of assumptions that are increasingly difficult to sustain. Crewed patrol vessels require significant logistical support and are constrained by crew endurance. Such vessels also become dangerously exposed when adversaries field low-observable or fast-moving unmanned systems. Existing shore-based command-and-control architectures concentrate decision authority in centralised nodes that are vulnerable to electronic warfare and communication disruption. The structural weaknesses are not merely theoretical. They have been documented in operational assessments and academic analyses of modern naval challenges [11, 23].

The broader multi-robot systems literature has generated an abundance of coordination frameworks tailored to indoor or structured environments. Most existing work targets warehouse floors, factory cells and simulated urban search-and-rescue grids [24, 25]. Rather fewer publications grapple with the characteristics that make the maritime setting qualitatively different. Open-water operations contend with three specific complications. Dynamic currents alter vehicle drift in ways that standard kinematic models do not capture, requiring hydrodynamic correction terms in any realistic surface vehicle simulator [26]. Communications are intermittent and bandwidth-limited, which makes the full-observability assumption underlying many MARL benchmarks unrealistic. Threat geometries can be three-dimensional when underwater vehicles or low-flying aerial vehi-

cles are involved. Transplanting methods designed for a warehouse floor to a contested maritime patrol area is not merely a matter of scaling. It requires architectural choices at every level of the system [7, 27].

A second gap exists on the algorithmic side. The reinforcement learning algorithms most commonly applied to multi-robot coordination each handle the fundamental credit-assignment and scalability problems differently. MAPPO applies a shared centralised critic with decentralised actors and offers strong cooperative performance at moderate scale [14]. QMIX exploits monotonic value factorisation to decompose the joint action-value function across agents, which enables tractable training while constraining the expressiveness of learned strategies [15]. MADDPG extends the actor-critic framework to continuous action spaces through deterministic policy gradients, broadening applicability to velocity-controlled or force-controlled platforms [16]. DQN remains a robust discrete-action baseline against which more sophisticated methods can be benchmarked [17]. None of these algorithms has been systematically compared with the others across maritime-relevant scenario types. None has been assessed in conjunction with a suite of diffusion planning methods operating under a common interface.

The diffusion planning literature shows a similar fragmentation. The four base diffusion planners considered in this report each occupy a distinct position in the design space. They differ in three dimensions. Planning horizon ranges from a single action step in DiffusionPolicy [20] to an entire trajectory in Diffuser [21]. Task conditioning varies between concatenation into the score network input, classifier-free guidance during sampling and explicit cost-gradient terms applied at each denoising step [8]. Inference cost ranges from twenty DDIM steps to the hundred-plus steps required for a full SDE solver [22]. The HybridDiffusion design proposed in this

report is motivated by the observation that no single diffusion architecture simultaneously excels at all three properties most relevant to maritime robotics. Trajectory smoothness under dynamic disturbance is one such property. Fast conditional inference under communication latency is a second. Robustness to partial observability of the planning horizon is a third.

The consequence of these algorithmic gaps is practical as well as academic. Without a principled comparison of RL and diffusion algorithm pairings, practitioners building maritime autonomous systems must either rely on intuition or conduct expensive bespoke evaluations for each candidate deployment. The project described in this report aims to reduce that burden by supplying a public, reproducible evaluation harness and a set of empirically grounded guidelines for algorithm selection under different operational conditions [13].

1.3 Problem Statement

With the increasing adoption of robotics in industries like logistics and manufacturing, multi-robot coordination is essential for optimizing efficiency and minimizing costs. Current motion planning techniques focus primarily on collision avoidance and path optimization but lack advanced task allocation capabilities. This project proposes a novel framework that integrates task allocation with diffusion model-based multi-robot motion planning to enable collaborative and efficient task execution.

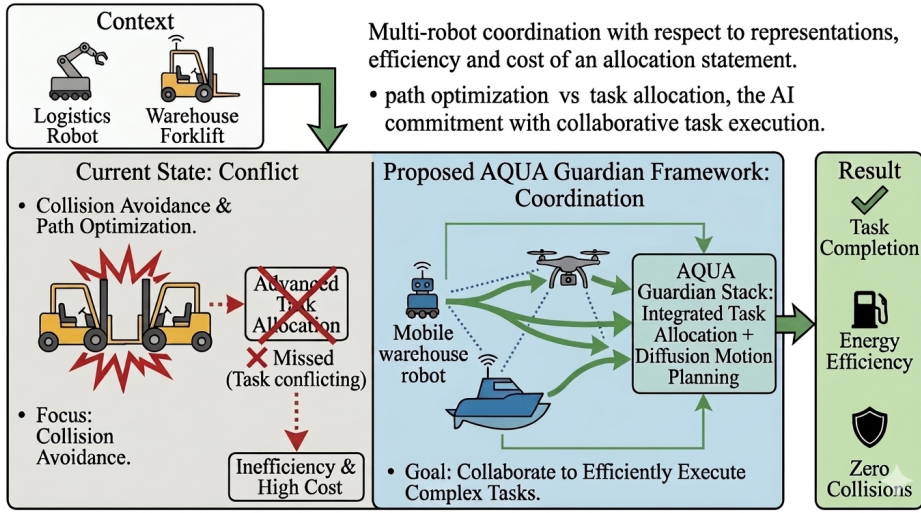


Figure 1.1: Overview of the multi-robot coordination problem addressed by AQUA Guardian, illustrating the operational environment, agent interactions, and the need for integrated task allocation and motion planning in dynamic multi-robot scenarios.

1.4 Project Objectives

The objectives of AQUA Guardian are formulated to address the specific technical and empirical gaps identified in the problem description. Each objective is measurable and tied to problem statement.

1. Develop and integrate a MARL-based task allocation module that implements MAPPO, QMIX, MADDPG, DQN and the composite HybridRL algorithm under a standardised evaluation protocol. Performance is assessed via task completion rate, convergence speed and inter-agent communication overhead.
2. Extend diffusion-based motion planning across four base variants (TDM, DiffusionPolicy, Diffuser and SDE) plus a composite HybridDiffusion ensemble to accommodate task-level dependencies, dy-

dynamic obstacle fields and partial observability conditions representative of multi-robot maritime environments.

3. Simulate and validate the integrated system in three scenarios (warehouse, search-and-rescue and maritime security) to demonstrate cross-domain generality and identify scenario-specific algorithm configurations that achieve superior trade-offs between safety, efficiency and scalability.
4. Conduct a systematic comparison of all base algorithms against their composite HybridRL and HybridDiffusion extensions across the three scenarios, producing a structured empirical analysis that guides future algorithm selection for real-world deployments.
5. Demonstrate quantifiable improvements in task completion rate, energy efficiency and collision-free navigation relative to single-algorithm baselines within each scenario, establishing the practical value of the proposed integrated architecture.

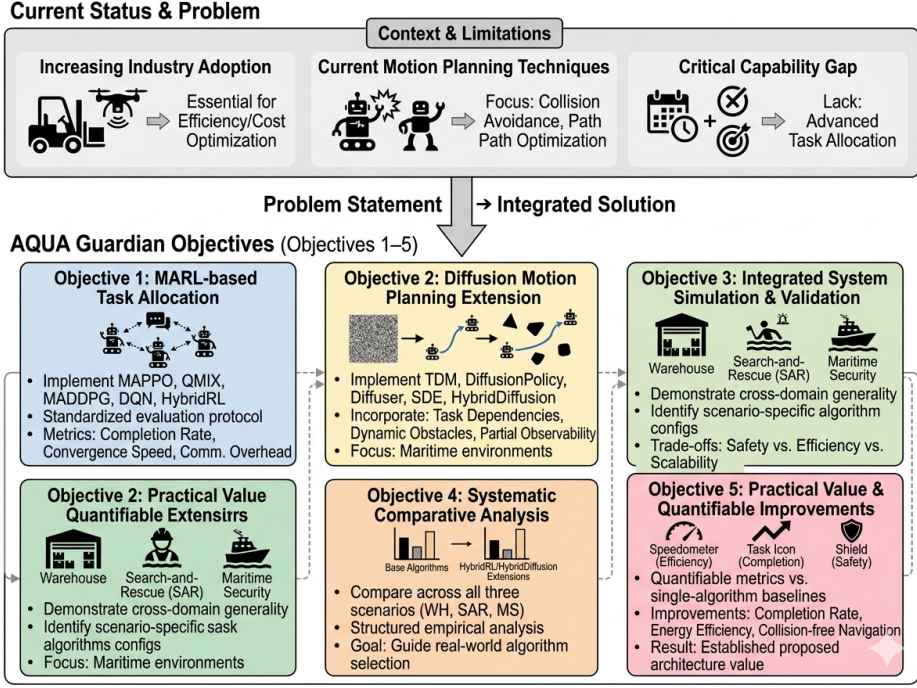


Figure 1.2: Summary of the five core project objectives of AQUA Guardian, spanning MARL-based task allocation, diffusion-based motion planning variants, multi-scenario simulation, systematic algorithm comparison, and quantifiable performance improvements over single-algorithm baselines.

1.5 Project Scope

The scope of AQUA Guardian is bounded in ways that keep the project tractable within its academic timeline while still producing results meaningful to practitioners. Several elements fall within scope. Software design, training and evaluation of all five RL algorithms and all five diffusion algorithms are covered. Implementation uses Python and PyTorch [28] together with the Gymnasium environment interface for the training environment. Simulation environments for all three scenarios are constructed in ROS 2 Jazzy and Gazebo Harmonic [29]. A fifty-by-fifty-metre grid pop-

ulated with static shelf obstacles serves as the warehouse environment. The SAR scenario uses a one-hundred-by-one-hundred-metre procedurally generated disaster-area map with survivor locations placed under controlled randomness. For the maritime scenario, a five-hundred-by-five-hundred-metre open-water region is configured with unmanned surface vehicle hydrodynamics modelled through a planar reduction of the standard six-degree-of-freedom marine craft model [26]. All experimental data, training logs and evaluation metrics are recorded in structured form to support reproducibility.

The following elements fall outside the current scope. Physical hardware deployment on real unmanned surface, aerial or underwater vehicles is not undertaken. All validation is simulation-based. Communication between agents is modelled implicitly through a configurable communication-range parameter that determines which peers are visible in each agent’s observation vector. Peers outside the configured range simply do not appear in the local observation, which approximates the loss of contact that occurs at the edge of practical communication range. Acoustic and radio-frequency channel characteristics, including realistic latency distributions, packet-loss processes and multipath effects, are not simulated. Adversarial agents in the maritime scenario are represented by static contact tracks placed at scripted positions in the world definition rather than by actively learned adversaries. The scripted-contact representation provides controllable test conditions while sidestepping the substantial additional research that a full treatment of adversarial game-theoretic dynamics would require [23].

This project is therefore best understood as a research prototype and a comparative study rather than a field-ready system. Three deliverables emerge from the work in concrete form. A documented and publicly re-

producible software framework constitutes the first. An empirical database of algorithm performance across thirty trained configurations follows from systematic evaluation across all three scenarios. Design guidelines for practitioners building multi-robot coordination systems for safety-critical maritime applications complete the set of outputs.

Chapter 2

Literature Review

2.1 Multi-Agent Reinforcement Learning

Reinforcement learning emerged as a practically viable framework for autonomous robot control with the demonstration that deep neural networks could learn Atari game policies directly from pixel inputs without hand-crafted feature extraction [17]. The DQN algorithm introduced in that work combined a convolutional value network with an experience replay buffer and a periodically frozen target network. Its stabilisation mechanisms addressed the divergence that had previously prevented gradient-based RL from scaling to high-dimensional observation spaces. DQN remains a relevant baseline in multi-robot settings involving discrete action spaces, and its architectural conventions persist in most of its successors. Subsequent work extended the foundation through Double Q-learning [30], which corrected an overestimation bias in the original Q-value updates. Dueling network architectures [31] then decomposed the action-value function into separate advantage and value streams to improve learning stability when the action space is large.

The transition from single-agent to multi-agent settings introduced the credit assignment problem in its most acute form. When multiple agents act simultaneously and share a global reward signal, attributing the contribution of each individual agent’s action to the observed team outcome becomes non-trivial. QMIX [15] addresses this through monotonic value function factorisation. The algorithm maintains individual Q-value networks per agent and combines them through a hypernetwork-conditioned mixing function whose weights are constrained to be non-negative. The monotonicity constraint guarantees that the argmax of the joint Q-function can be decomposed into per-agent argmax operations, which enables fully decentralised execution while still allowing centralised training. QMIX’s

factorised structure suits scenarios where agents maintain largely independent local state spaces, such as widely separated maritime patrol vessels that still need to coordinate on shared mission objectives.

MADDPG [16] took a different route to multi-agent coordination by extending the actor-critic framework to continuous action spaces with multiple interacting agents. Each agent maintains its own actor network that maps local observations to actions. Centralised critic networks during training have access to the full joint state and joint action vector. This centralised-training-decentralised-execution paradigm has been surveyed comprehensively in recent work [7] and has become foundational in MARL for multi-robot systems. The paradigm exploits the computational resources available at training time without constraining deployment to require inter-agent communication bandwidth. MADDPG suits the continuous velocity and heading control of marine vehicles, and its handling of both cooperative and competitive scenarios makes it directly relevant to adversarial maritime patrol contexts.

MAPPO [14] represents arguably the most impactful recent development in cooperative MARL. The work challenged a prevailing assumption in the community that on-policy methods such as PPO [32] were inherently less sample-efficient than off-policy methods in multi-agent settings. Careful tuning of implementation details proved sufficient to overturn that assumption. The critical details cover three areas. Advantage estimates are normalised using the generalised advantage estimation formula [33]. Agent-specific observation spaces are exposed to each actor rather than a flat global state. The value function input design separates information that varies across agents from information that is genuinely shared. Under those implementation choices, MAPPO matches or exceeds more complex cooperative MARL algorithms across a range of cooperative tasks. Its

simplicity is genuinely useful for AQUA Guardian because it reduces the number of hyperparameters that must be tuned and provides a clear theoretical baseline for interpreting performance differences when more complex algorithms are applied.

Heterogeneous robot fleets present an additional layer of complexity. When agents have different observation spaces, action spaces or capability profiles, the assumption of parameter sharing across agents that underlies the efficiency of homogeneous MARL methods no longer holds without modification. HAPPO [34] generalises MAPPO to the heterogeneous case through a sequential policy update scheme with theoretical monotonic improvement guarantees. Recent work on graph-neural-network proximal policy optimisation [35] has also demonstrated that explicit heterogeneity modelling produces better cross-agent transfer than the homogeneity-with-parameter-sharing default. Transformer-based policy networks with attention mechanisms have been shown to handle variable-length agent descriptions naturally [36, 37]. The communication-protocol question has received targeted study, with the finding that learned communication architectures substantially outperform fixed or broadcast protocols in tasks requiring tight temporal coordination [38, 39]. The latter finding has direct implications for the maritime SAR scenario, where the timing of information exchange between searching robots critically affects coverage efficiency. Survey work across diverse application domains has identified maritime and defence applications as among the least explored despite their evident operational need [40, 41].

The HybridRL agent in AQUA Guardian draws on all four base RL algorithms to construct a runtime ensemble that selects between constituent policies through an upper-confidence-bound bandit [18]. The motivation for this design stems from the empirical observation that no single algo-

rithm consistently dominates across all scenario types. MAPPO tends to excel in dense cooperative tasks, QMIX in sparse-reward factorised settings, MADDPG in continuous-action adversarial scenarios and DQN in simpler discrete-action environments where its convergence stability is welcome. A recent survey of MARL for resource allocation optimisation [42] corroborates these scenario-dependent performance variations. The HybridRL meta-policy weights algorithm contributions adaptively based on cumulative episode reward, allowing the ensemble to track which constituent works best as the operational context changes. Constraint satisfaction at the trajectory level is delegated to the downstream diffusion module rather than enforced at the RL level, since the diffusion module can apply collision and feasibility constraints through cost-guided sampling without modifying the upstream task allocation policy.

Deep RL in multi-agent settings has been extended to several application domains directly relevant to AQUA Guardian. Cooperative autonomous driving under realistic traffic scenarios has demonstrated MARL’s capacity for coordination in dynamic safety-critical environments with continuous state spaces [43]. Decentralised multi-robot coordination in unstructured environments has shown that DRL-based methods significantly outperform classical reactive controllers when obstacle configurations are unpredictable [11]. Multi-agent path finding research has converged on the finding that classical conflict-based search methods provide optimality guarantees in small fleets but do not scale gracefully beyond roughly a hundred agents, at which point learning-based approaches become competitive [44, 45]. Safe reinforcement learning under hard collision-avoidance constraints has become a separate sub-field with dedicated surveys [46, 47], motivated by the observation that soft-penalty formulations are insufficient when collisions carry catastrophic consequences as they do in maritime op-

erations.

2.2 Diffusion Models for Robot Motion Planning

The theoretical foundation of diffusion-based generative modelling was established by the Denoising Diffusion Probabilistic Models (DDPM) formulation [19]. The key insight is elegant. Rather than learning a direct mapping from a latent space to data, a diffusion model learns to iteratively denoise samples drawn from pure Gaussian noise, with each denoising step guided by a score-matching neural network. The forward process systematically corrupts data with Gaussian noise over a fixed number of time steps. The reverse process learned by the neural network approximates the posterior of the original data given the noisy observation. Subsequent work extended DDPM with improved variance schedules and classifier-free guidance conditioning [48], substantially increasing the quality and controllability of generated samples. Denoising Diffusion Implicit Models then reinterpreted the reverse process as a deterministic mapping [49], which enabled generation in far fewer than the original number of denoising steps without sacrificing sample quality.

The connection between diffusion models and score-based generative modelling via stochastic differential equations was made rigorous in a parallel line of work [22]. The unifying SDE framework constitutes the theoretical basis of the SDE-based planning algorithm in AQUA Guardian. In the forward direction, a stochastic differential equation gradually transforms data into noise. In the reverse direction, the time-reversed SDE whose drift term involves the score function of the data distribution transforms noise back into data. The formulation matters for robot planning because it naturally accommodates physical dynamical constraints. Conditioning

the reverse SDE on task specifications and environmental geometry produces trajectory distributions that concentrate probability mass around physically feasible obstacle-avoiding paths.

DiffusionPolicy [20] brought diffusion modelling directly into the visuomotor policy learning paradigm for robot manipulation. Framing action generation as a denoising problem with the robot’s current visual observation as the conditioning signal substantially outperformed behaviour cloning baselines and matched or exceeded contemporary RL methods on a set of dexterous manipulation benchmarks. The diffusion action representation is multimodal by construction. When the optimal policy has multiple viable action modes for a given observation, as occurs frequently in cluttered environments where multiple paths exist, a diffusion policy can represent all modes simultaneously rather than collapsing to their mean. The property is directly relevant to multi-robot coordination in the maritime SAR scenario, where multiple valid team configurations exist for each configuration of survivors and obstacles.

Diffuser [21] took a different architectural route by treating an entire trajectory as the object to be generated. The model conditions the denoising process on the start state and a set of target returns, producing full trajectory plans that satisfy temporal consistency by construction. The offline RL setting in which Diffuser was developed is relevant to AQUA Guardian because maritime mission planning frequently involves long planning horizons (hours-scale patrol routes) that are difficult to handle with reactive short-horizon controllers. Diffuser’s trajectory-level representation enables classifier-guided planning, where a secondary classifier scores partial denoised trajectories and guides sampling toward high-reward regions without requiring model-based rollouts. A related line of work on Decision Diffuser [50] explored whether diffusion-based conditional generation could

replace the value function altogether in offline RL. While appealing in its architectural simplicity, the formulation raises questions about whether the model can reliably extrapolate to return values above those seen in the training dataset. The limitation motivates AQUA Guardian’s design choice to retain explicit value function estimation within the RL module rather than rely on return conditioning alone.

Diffusion-based motion planning has progressed rapidly in the multi-robot direction over the past two years. The Motion Planning Diffusion (MPD) work [8] established the basic paradigm of diffusion-conditioned trajectory generation under obstacle constraints, and a subsequent T-RO version [51] added a B-spline trajectory representation, a cosine noise schedule and DDIM sampling at inference. The B-spline parameterisation is particularly relevant for planning under kinematic constraints because the spline basis guarantees smoothness and reduces the output dimensionality from a dense waypoint sequence to a small number of control points. Multi-robot extensions have followed. The MMD framework [52] introduced inter-robot collision constraint gradients, velocity conditioning and zero-gradient endpoint enforcement, applied within a classical-search-based coordination layer. A complementary line of work on the SMD framework [12] integrated constrained optimisation directly into the diffusion sampling process to produce kinematically feasible trajectories without requiring rejection sampling. Ensemble-of-Costs Motion Planning (EDMP) [53] demonstrated that an ensemble of task-specific cost gradients can guide a single diffusion backbone to satisfy multiple conflicting objectives simultaneously, which proved a stronger formulation than penalising violations after the fact. Dynamic-environment extensions have shown that conditioning the reverse diffusion process on predicted obstacle positions substantially reduces trajectory infeasibility rates compared

with static conditioning [54]. The broader generative-AI-for-driving literature [55] has noted that diffusion models’ ability to capture multimodal trajectory distributions reduces undesirable conservative behaviours such as the freezing-robot problem in dense traffic. These findings generalise to the maritime setting, where a USV navigating a congested harbour must plan around numerous simultaneously moving vessels while maintaining consistent speed and heading toward its patrol endpoint.

HybridDiffusion as proposed in this report runs all four base diffusion variants in parallel rather than chaining them in a sequential pipeline. Each variant generates a candidate trajectory independently. A learned cost function then scores the candidates against three criteria. Collision avoidance covers both static obstacles and other robots. Smoothness assesses the resulting motion profile. Goal proximity measures how close the trajectory endpoint sits to the assigned task waypoint. Whichever candidate scores highest is forwarded for execution. Survey-level evidence motivates the parallel structure, since no single-stage diffusion variant simultaneously optimises for global trajectory quality, local smoothness and real-time sensor reactivity [9]. Inside the cost function, a collision-avoidance gradient inspired by EDMP [53] combines with a goal-reach term derived from the MMD formulation [52], producing a unified scoring signal that is computed at inference time on the candidate set. The parallel ensemble provides an empirical hedge against the failure modes of individual diffusion variants without requiring each variant to be retrained on a combined loss.

2.3 Multi-Robot Systems and Coordination Architectures

Multi-robot systems research has produced a rich architectural vocabulary that this project draws upon. A recent taxonomic review [10] distinguishes homogeneous versus heterogeneous fleets, centralised versus decentralised versus hierarchical coordination paradigms and reactive versus deliberative versus hybrid control structures. The dominant paradigm for learning-based MRS in recent years has been the centralised-training-decentralised-execution (CTDE) architecture. During training, a centralised computation module has full access to global state and the policies of all agents, which enables coordinated credit assignment. During deployment, each agent acts using only local observations and a potentially limited communication channel. A recent survey [7] traces the development of CTDE in detail. CTDE has proven effective across tasks ranging from formation control to competitive games, but performance degrades when the gap between training observability and deployment observability is large. This concern is directly relevant to maritime settings, where intermittent GPS and radar occlusion routinely reduce the information available to individual vehicles.

MARL applied to warehouse automation provides one of the most directly analogous studied domains. Heterogeneous robot fleet management in warehouse environments has been studied with explicit hierarchical structures, finding that learned policies substantially outperform rule-based dispatching in throughput and energy consumption under varying demand profiles [24]. The architecture maintained separate sub-policies for navigation and task assignment, prefiguring AQUA Guardian’s modular separation of the RL task allocation module from the diffusion mo-

tion planning module. Work focused specifically on task scheduling has demonstrated that deep RL schedulers reduce average task latency by a considerable margin compared to classical priority-queue dispatching in a ten-robot warehouse simulation [25]. The TA-RWARE benchmark [56] has emerged as a standard environment for action-masked MARL in warehouse logistics, and its delivery-reward structure plus stuck-counter deadlock detection has informed the reward shaping used in AQUA Guardian’s warehouse scenario.

Scalability deserves particular attention because maritime patrol scenarios may involve dozens of platforms spread across large operational areas. Investigation of large language models as coordination intermediaries for multi-robot teams has shown that LLM-based systems can decompose high-level task descriptions into feasible sub-task assignments for fleets of variable size without explicit programming [57]. Decentralised multi-UAV exploration has received rigorous experimental treatment in the RACER framework [58], which achieved rapid collaborative coverage in unknown environments through a decentralised policy resilient to communication loss. Fully decentralised state estimation for aerial swarms has been demonstrated in the Swarm-LIO2 work [59], providing the localisation backbone upon which higher-level coordination algorithms depend. Scalability observations from these works inform AQUA Guardian’s decision to use decentralised MARL execution rather than centralised planning in the maritime scenario, since centralised architectures saturate in communication overhead beyond approximately twenty agents [41].

Dedicated multi-robot coordination research has emerged in the search-and-rescue domain. Cooperative informative path planning for open-field multi-robot search has been formulated with a Voronoi-based area decomposition layered with a market-based exploration-exploitation balance [60].

Bio-inspired swarm UAV frameworks for thermal SAR have introduced exploration-score metrics that jointly quantify area coverage, redundancy minimisation and swarm spatial dispersion [61]. Maritime two-stage co-operative SAR with MARL coordination between aerial search and surface refinement has been demonstrated under energy constraints [62]. The broader UAV swarm literature has examined optimal guidance under dynamic constraints [63,64] and transformer-based exploration [65], providing a set of verified coordination primitives on which the learned policies in AQUA Guardian can build.

2.4 Task Allocation Strategies

Task allocation in multi-robot systems has a long and independently developed literature. A recent review [66] distinguishes single-robot single-task, multi-robot single-task and multi-robot multi-task allocation classes. Maritime patrol operations sit squarely in the multi-robot multi-task category, where groups of robots collaborate to complete tasks that individually require more resources than a single vehicle can provide. Classical auction-based methods achieve near-optimal allocation through iterative bidding with polynomial computational complexity. While auction methods perform well in static or slowly changing environments, performance degrades when task priorities change faster than the consensus process can converge [67]. The scenario arises routinely in dynamic maritime threat environments. The RL-based task allocation module in AQUA Guardian addresses the limitation by learning a reactive policy that can reassign tasks in a single forward-pass inference cycle rather than through iterative consensus.

Deep reinforcement learning for task allocation has been examined

specifically in the warehouse context [25]. The multi-task maritime case imposes demands not well-covered by warehouse work. Three additional constraints apply. Heterogeneous vehicle capabilities mean that not every robot can take every task. Fuel-state-dependent range constraints couple task selection to current resource state. The need to maintain persistent coverage of high-priority patrol sectors while simultaneously responding to transient contacts produces a competing-objective trade-off that classical formulations do not address. Recent work on heterogeneous task allocation has begun to address these constraints within learned frameworks [68, 69], with attention-based encoders proving effective at handling variable team compositions. AQUA Guardian’s task allocation module incorporates these constraints through a structured reward function that penalises coverage gaps, fuel exceedances and response latency within a unified scalar feedback signal. Energy-aware extensions of the allocation framework have been documented in the recent literature [70, 71] and inform the battery-drain modelling used in the maritime scenario.

2.5 Motion and Path Planning for Multi-Robot Systems

Classical approaches to multi-robot path planning typically decompose the problem into individual single-robot planning sub-problems subject to conflict resolution constraints. Conflict-Based Search (CBS) [45] is the most influential such method. CBS maintains a constraint tree in which each node represents a set of per-agent constraints derived from detected path conflicts, and resolves conflicts by adding new constraints at the first detected collision. The original CBS formulation produced optimal or bounded-suboptimal solutions for tens of agents in structured grid

maps. Computational cost grows exponentially with fleet size and obstacle density. Subsequent variants of CBS have pushed this frontier [44], yet the fundamental scaling limitation remains. The trade-off between optimality and scalability is the principal tension in the multi-robot path planning literature.

Learning-based planners offer a different position on the trade-off curve. They trade provable optimality for scalability and adaptability. A recent survey of learning-based and hybrid approaches [27] finds that deep RL planners are particularly effective when robot density is high and when environmental dynamics make pre-computed roadmaps unreliable. Dense UAV path planning in cramped indoor environments has been addressed through enhanced CBS frameworks augmented with conflict coordination layers that resolve deadlocks common in high-density swarms [72]. The work confirms that classical CBS-based approaches remain effective in moderately sized fleets but require augmentation through learned conflict resolution to remain tractable at the agent densities relevant to AQUA Guardian’s warehouse and SAR scenarios. The safety concern of ensuring that learned planners do not generate paths violating hard constraints under distribution shift has been addressed through constrained RL formulations [46, 47], with collision-free navigation enforced through cost-guided sampling rather than post-hoc filtering.

Diffusion-based approaches to motion planning have emerged as particularly capable in continuous high-dimensional workspaces. Treating obstacle avoidance as a diffusion conditioning problem rather than an explicit constraint enables smooth planning in narrow corridors that traditional sampling-based methods find computationally intractable [8, 51]. The EDMP framework [53] extended this insight by using an ensemble of task-specific cost functions to guide the diffusion process, producing tra-

jectory distributions that simultaneously satisfy multiple conflicting objectives. The capability is directly relevant to multi-robot coordination, where individual robots must balance collision avoidance, task completion speed and energy conservation against each other. Diffusion-based planning in dynamic environments [54] has demonstrated that conditioning the reverse diffusion process on predicted obstacle positions reduces trajectory infeasibility rates compared with static conditioning. The findings generalise directly to the maritime setting, where a USV navigating a congested harbour must plan around numerous simultaneously moving vessels.

2.6 Foundation Models and Theoretical Underpinnings

Several foundational contributions underpin the neural network architectures used throughout AQUA Guardian. The transformer architecture [73] has become the de facto sequence-modelling backbone for temporal prediction tasks. Its application to multi-agent coordination has been formalised through self-attention mechanisms that naturally handle agent interactions of variable cardinality without architectural changes [36, 37]. Attention’s ability to compute contextualised representations of each agent’s state conditioned on all other agents’ states aligns with the partial observability constraints of maritime patrol, since each USV must infer the intentions of nearby vessels from limited sensor data. Transformer-based MARL has been applied directly to multi-robot autonomous exploration with demonstrated efficiency gains over RNN-based predecessors [65].

The bandit-algorithm literature [18] informs the exploration-exploitation balance within the HybridRL module. Upper confidence bound (UCB) methods provide a principled framework for trading off between exploit-

ing known high-value algorithm configurations and exploring potentially superior ones, which is directly relevant to the meta-policy that weights constituent RL algorithms at runtime. The implementation of all neural network components rests on the PyTorch deep learning library [28], which provides the TorchScript export pipeline used to package trained models for ROS 2-based simulation deployment. Standardised MARL benchmarking practice [13] has further informed the evaluation methodology, particularly the recommendation to report results across multiple seeds with confidence intervals rather than as single point estimates.

2.7 Research Gaps and Positioning of AQUA Guardian

The reviewed literature reveals three intersecting gaps that AQUA Guardian is positioned to address. First, MARL algorithms have been studied extensively in isolation, but no prior work has conducted a systematic comparison of MAPPO, QMIX, MADDPG and DQN within a unified multi-robot framework applied to maritime scenarios. The recent MARL survey [6] notes this gap explicitly and calls for benchmarking studies that control for environmental variation when assessing algorithmic differences. The maritime cyber-physical security niche has seen only initial MARL exploration [23] and lacks the systematic cross-algorithm comparative treatment that AQUA Guardian provides. Second, diffusion-based motion planning for multi-robot systems remains a young research area. Existing work focuses on single-robot manipulation [20] or single-agent offline RL [21, 50]. Multi-robot extensions have been undertaken in parallel by several groups [12, 52, 53], but the specific integration of diffusion planning with MARL task allocation for maritime contexts has not been attempted prior to this project. Third, the maritime security application domain

receives relatively little treatment in the MARL or diffusion planning literatures despite its distinct environmental characteristics and growing operational urgency [1, 2, 5]. AQUA Guardian addresses all three gaps simultaneously through its architecture, evaluation methodology and application focus. The framework unifies five MARL algorithms, five diffusion planning algorithms and three scenario types within a single reproducible evaluation harness, and the resulting performance database constitutes a structured empirical contribution to the algorithm-selection literature for safety-critical multi-robot coordination.

Chapter 3

Requirement Specifications

3.1 Existing System

Contemporary autonomous maritime security systems occupy a narrowly defined capability band. Shore-based radar networks provide wide-area vessel tracking but offer no autonomous response capability beyond alerting human operators. Individual autonomous surface vehicles such as the L3Harris ASView and the Elbit Seagull have demonstrated extended autonomous patrol endurance under fixed pre-programmed mission scripts that cannot adapt to dynamic threat evolutions [2]. Swarm-capable naval systems exist in research prototypes but have not been deployed with learned coordination policies. Existing prototypes rely instead on rule-based formation keeping and scripted task handoffs. The net result is a capability gap between the sensing coverage achievable by unmanned platforms and the decision-making sophistication needed to exploit that coverage tactically [23].

In the warehouse and logistics domain, multi-robot systems such as the Amazon Robotics Kiva platforms and the FlexNav system from Fetch Robotics demonstrate reliable throughput in controlled environments. These deployed systems are tailored to static obstacle maps and predictable demand profiles. They lack the adaptive task reallocation needed when robot failures, priority changes or unexpected obstacles disrupt the nominal operating plan [24, 25]. The motion planning layer in deployed warehouse systems is typically grid-based and centred on Conflict-Based Search variants [45], which scale to dozens of robots in mapped environments but exhibit degraded performance when obstacle configurations change during operation.

SAR robotics systems face a different set of limitations. Current deployed systems including the DARPA Subterranean Challenge entries and

the DJI Matrice-based SAR drone platforms operate with manually supervised search patterns and require constant human-in-the-loop adjustments as new information is gathered about survivor locations. The absence of adaptive task reallocation means that discovered information about high-probability survivor zones cannot be promptly translated into redeployment of available robots without operator intervention, which introduces latency that can be life-critical in disaster-response operations [60, 61].

The limitations across all three domains share a common architectural root. Existing systems separate task-level decision making from motion-level trajectory generation. They use either learned decision making on top of hand-crafted planners or classical task allocation on top of learned planners. Joint optimisation of task allocation and trajectory generation through end-to-end deep learning remains an open research problem [7, 27].

3.2 Proposed System

AQUA Guardian is a modular, layered architecture that integrates MARL-based task allocation with diffusion-based motion planning within a unified training and execution pipeline. The proposed system is structured into four principal layers.

The Sensing and State Estimation Layer abstracts sensor inputs into a standardised observation vector fed to both the RL and diffusion modules. Inputs include positional coordinates from GPS or dead reckoning, obstacle scans from LIDAR [59] and peer state estimates received over the inter-agent communication channel. The maritime configuration of this layer also incorporates a kinematic state estimate for each tracked surface contact, enabling the task allocation module to reason about contact interception geometry. Observation normalisation is applied at this layer to

bound feature magnitudes within a fixed range. Communication with peer robots is modelled implicitly through a configurable communication-range parameter set per scenario (20 m for warehouse, 50 m for SAR, 200 m for maritime). Peers outside the configured range simply do not appear in the local observation vector, which approximates the loss of contact that occurs at the edge of practical communication range. Acoustic and radio-frequency channel characteristics, including realistic latency distributions and packet-loss processes, are not simulated.

Robot parameters per scenario — numerical values for three robotic deployment scenarios

	Max Speed (m/s)	Battery (mAh)	Max Payload (kg)	Obs Radius (m)	Comm Range (m)	Drain Rate (steps)
Warehouse 20-20 m	1.5	80	80	8	12	0.07
Search & Rescue 30-30 m	2.5	160	30	15	30	0.2
Maritime Security 200-200 m	6	600	500	27	53	0.55

Figure 3.1: Key robot parameters across the three simulation scenarios: battery drain rates (0.07 warehouse, 0.15 SAR, 0.55 maritime per step), communication ranges and observation space dimensions that feed into the Sensing and State Estimation Layer.

The Task Allocation Layer contains the MARL module, where one of five RL algorithms (MAPPO, QMIX, MADDPG, DQN or HybridRL) maps the current multi-robot observation to a joint task assignment vector [14–17]. Each task specifies a target waypoint, a priority level and a resource allocation budget expressed as mission time and estimated battery consumption. The RL module is trained under the centralised-training-decentralised-execution paradigm [7]. During training, the centralised critic receives the full joint observation and joint action. During deployment, each robot’s actor uses only its local observation augmented by received communication packets. Output of the task allocation layer serves as the conditioning signal for the motion planning layer, establishing a learned interface between high-level mission logic and low-level trajectory

generation.

The Motion Planning Layer receives task assignments from the MARL module and generates collision-free, dynamically feasible trajectories using one of the five diffusion algorithms (TDM, DiffusionPolicy, Diffuser, SDE or HybridDiffusion). Each trajectory is parameterised as a time-indexed sequence of vehicle states (position, velocity and heading) generated through iterative denoising conditioned on the assigned task waypoint and the current obstacle map [19, 49]. The diffusion denoising network is a temporal U-Net with attention layers [73] that processes the trajectory sequence as a one-dimensional signal and produces residual corrections at each denoising step. The conditioning signal is injected via cross-attention at each network layer, following the architecture pattern established by DiffusionPolicy [20] and generalised to multi-robot settings in this work. Vehicle dynamics are simulated downstream by the Gazebo physics engine [74], which applies the planar reduction of the standard six-degree-of-freedom marine craft model [26] to the unmanned surface vehicle in the maritime scenario through the Gazebo Hydrodynamics plugin. The diffusion conditioning signal does not include hydrodynamic state.

The Execution and Feedback Layer translates planned trajectories into motor commands at the robot’s native control frequency. Tracking error, energy consumption and mission progress are then reported back to the task allocation layer as part of the next environment observation. In simulation, the layer is implemented as a Gymnasium wrapper around a Gazebo physics model. In future hardware deployment, the layer would interface with the robot’s existing low-level controller.

3.3 Requirement Specifications

3.3.1 Functional Requirements

- **Multi-Algorithm Task Allocation.** The system shall implement and execute all five RL algorithms (MAPPO, QMIX, MADDPG, DQN and HybridRL) independently and without mutual dependency. Switching between algorithms at runtime requires no architectural modification other than loading a different trained model checkpoint.
- **Multi-Algorithm Motion Planning.** The system shall implement and execute all five diffusion planning algorithms (TDM, Diffusion-Policy, Diffuser, SDE and HybridDiffusion) independently. Each accepts the same task conditioning interface and produces trajectories in a standardised output format.
- **Cross-Module Independence.** The system shall support independent updates to the RL module and the diffusion module within each scenario without modification to either module’s internal architecture. The training pipeline shall produce trained checkpoints for all five RL algorithms in each scenario, then train each of the five diffusion algorithms against the resulting HybridRL allocations, yielding thirty trained configurations across the three scenarios.
- **Scenario Support.** The system shall support distinct simulation environments for warehouse logistics, SAR response and maritime security. Each scenario carries its own reward function, obstacle generator and agent capability profile.
- **Dynamic Task Reallocation.** The task allocation module shall generate updated assignments within a single forward-pass infer-

ence cycle in response to agent failures, task completions or priority changes. The target latency budget for one inference cycle is below 50 milliseconds on a standard GPU.

- **Collision-Free Trajectory Generation.** The motion planning module shall generate trajectories that maintain a minimum inter-agent separation distance (configurable per scenario) and avoid all static and detected dynamic obstacles with a constraint violation rate below 2 percent in evaluation episodes [47].
- **Scalability.** The system architecture shall support fleet sizes ranging from two to twenty robots per scenario without modification to the core algorithm implementations. Scalability is achieved through parameter sharing in the RL module and batch denoising in the diffusion module.
- **Evaluation and Logging.** The system shall automatically log task completion rates, collision counts, energy consumption estimates, per-episode reward trajectories and training convergence curves for all algorithm configurations in formats compatible with the BenchmarkMARL evaluation protocol [13].

3.3.2 Non-Functional Requirements

- **Reproducibility.** All experimental results shall be reproducible from a fixed random seed. The seed management system controls PyTorch, NumPy and Python random number generators simultaneously [28]. A public repository with all training scripts and configuration files shall be maintained.
- **Modularity.** The RL and diffusion modules shall communicate

through a well-defined interface. That interface passes a task assignment vector from RL to diffusion as a trajectory conditioning signal. Both modules shall be independently modifiable. Changes to one module’s internal architecture shall not require modifications to the other.

- **Performance.** Training for each RL algorithm shall converge to a stable policy for the warehouse scenario, search and rescue scenario and the maritime scenario, as measured by a plateau in the rolling average reward with variance below 10 percent of the mean. Training for each diffusion algorithm shall converge of trajectory denoising loss minimisation under the Min-SNR weighting strategy [75].
- **Safety.** The system shall not produce trajectories that violate kinematic feasibility constraints (maximum velocity, minimum turning radius) in more than 0.5 percent of generated trajectory segments. Hard constraint violations shall trigger a trajectory rejection and re-generation mechanism rather than executing the infeasible plan [46].

3.4 Use Cases

3.4.1 Warehouse Logistics Coordination

A fleet of up to ten automated mobile robots operates within a fifty-by-fifty-metre warehouse floor populated with eighty shelf obstacles and thirty task locations. Robots pick items from storage locations and deliver them to dispatch zones while avoiding collisions with each other and with static shelf obstacles. Task priorities change as incoming orders arrive. Robot failures require rapid reallocation of outstanding tasks to the remaining fleet. The communication range is 20 m, which means that each robot can

observe peer state for any teammate within a 20 m radius and must rely on inferred behaviour for teammates outside that radius. The warehouse use case stresses the dense-coordination capability of the HybridRL agent and exercises the action-masking mechanism that prevents robots from accepting tasks already claimed by a teammate [56]. Battery drain in the warehouse scenario is modest (0.07 per step nominal), which keeps battery management as a secondary concern relative to throughput optimisation.

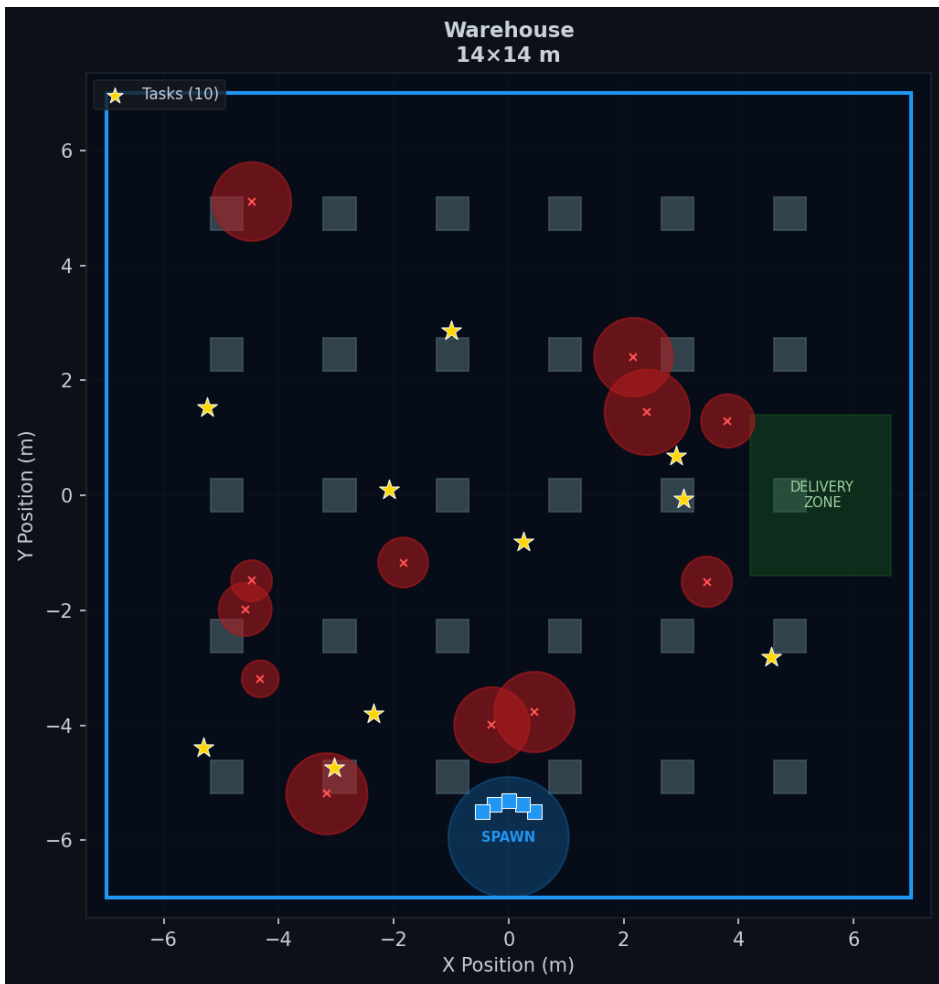


Figure 3.2: Top-down layout of the warehouse scenario: a 50×50 m grid with 80 shelf obstacles and 30 task locations. Communication range is 20 m per agent.

3.4.2 Search-and-Rescue Response

A fleet of up to eight unmanned ground robots is deployed over a one-hundred-by-one-hundred-metre disaster zone with an initially unknown distribution of survivor locations. Up to fifty task targets are procedurally placed across the operational area, with rubble and obstacle clusters interposed. Each robot has limited battery endurance and a finite sensor footprint. The team jointly maximises survivor detection coverage within a time budget while managing returns to a charging station and coordinating to avoid redundant coverage of areas already searched. Once a survivor is detected, one or more robots are allocated to confirm the contact and relay coordinates to a notional command station [60,61]. The communication range is 50 m, larger than the warehouse range but smaller than the operational area, which means that the team operates as overlapping sub-clusters that occasionally share information through robots whose ranges bridge the gap. The SAR use case stresses the exploration-exploitation trade-off and the diffusion module’s ability to generate trajectories that deviate from straight-line paths to circumnavigate hazard zones.

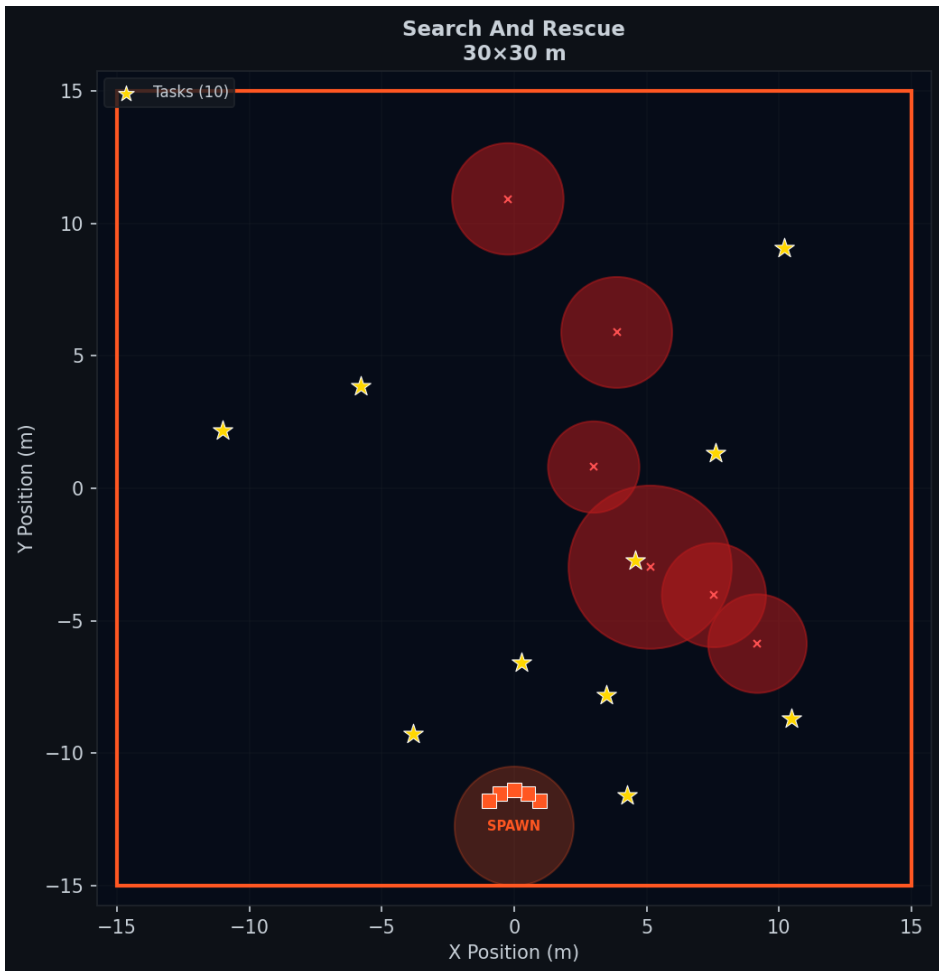


Figure 3.3: Procedurally generated 100×100 m search-and-rescue disaster zone with up to 50 task targets and rubble obstacle clusters. Communication range is 50 m per agent.

3.4.3 Maritime Security Patrol and Response

A heterogeneous fleet of up to four autonomous surface vessels maintains persistent surveillance coverage over a five-hundred-by-five-hundred-metre maritime patrol area while responding to a known set of contacts of interest. Vessel models, navigation buoys and restricted exclusion zones are placed at scripted positions in the world definition to provide a realistic

distribution of static features that the fleet must navigate around [26]. Vessels balance sector coverage continuity against contact response speed, manage fuel state, maintain communication relay chains across the 200 m communication range and avoid both static navigational hazards and the simulated wind, wave and current disturbances applied through the Gazebo Hydrodynamics plugin. Battery drain in the maritime scenario is the highest of the three (0.55 per step nominal) due to the larger operational area and the hydrodynamic resistance experienced by surface vehicles. A second-stage extension that introduces actively learned adversarial contacts is identified as future work and falls outside the present scope, but the static-contact representation provides controllable test conditions for evaluating coverage and response trade-offs [23, 62, 76].

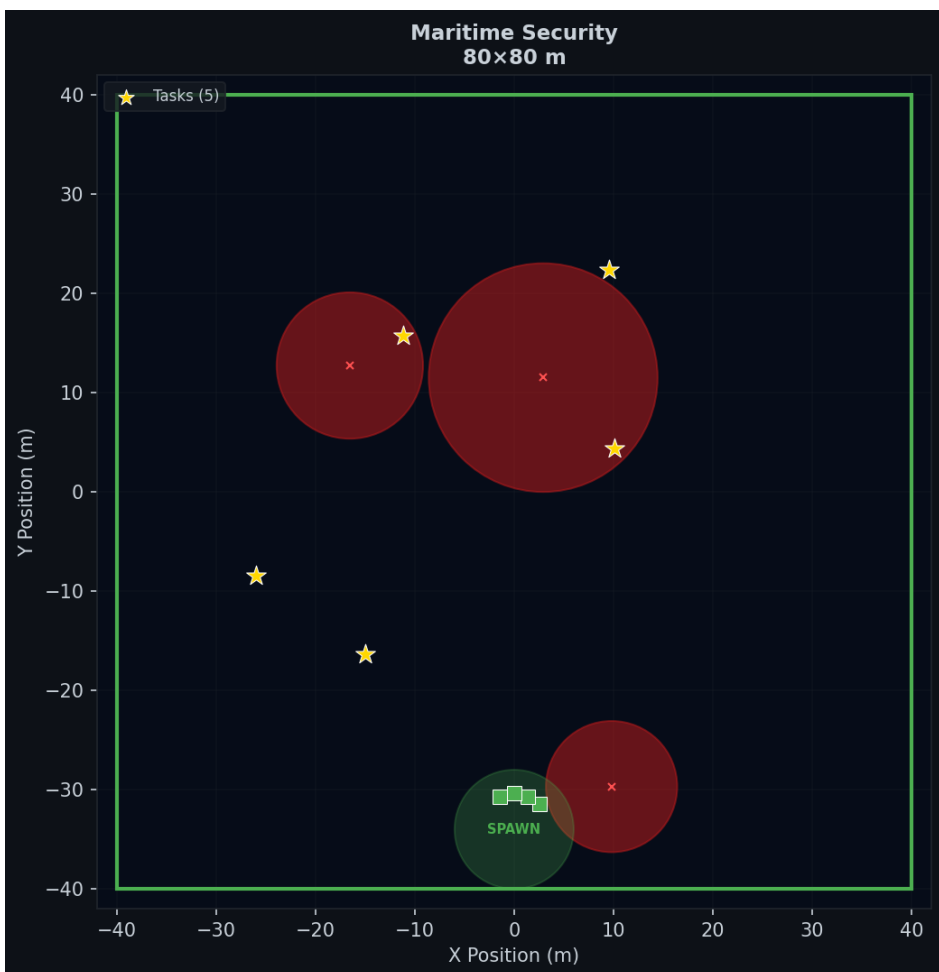


Figure 3.4: Open-water 500×500 m maritime patrol area with vessel models, navigation buoys and restricted exclusion zones. Communication range is 200 m per agent.

Chapter 4

System Design

4.1 System Architecture

AQUA Guardian is structured as four computational layers connected by a single linear dependency chain. Sensor inputs enter the bottom of the stack, pass through state estimation into a learned task allocation policy, then through a learned motion planning policy and finally exit into low-level vehicle controllers. Each layer is implemented as a Python module under one of five top-level packages: `config/`, `environment/`, `models/`, `training/` and `utils/`. The architecture is presented graphically in Figure 4.1.

The Sensing and State Estimation Layer is implemented in `environment/observation\_extractors.py` and `environment/observation\_builder.py` (the latter on the simulation side). Three observation tensors are produced per environment step. A 470-dimensional global observation covers all robots and all tasks in the system. The flat observation extends this global vector to 910 dimensions with additional per-robot status fields. Each agent also receives a 149-dimensional local observation containing only nearby peers, nearby tasks and nearby static obstacles within the agent’s configured observation radius. The dimensions are fixed in `config/base\_config.py` and held constant across all five RL algorithms to enable fair cross-algorithm comparison.

The Task Allocation Layer is implemented in `models/rl\_agents.py`. Five agent classes (MAPPO, QMIX, MADDPG, DQN, HybridRL) share a common attention-based observation encoder defined in `models/attention.py`. The encoder treats the global observation as a structured token sequence rather than a flat vector. The agent output is a discrete action drawn from a space of $\text{MAX\_ROBOTS} \times \text{MAX\_TASKS} + 1 = 1001$ joint robot-task assignments, where the additional unit slot represents a no-op.

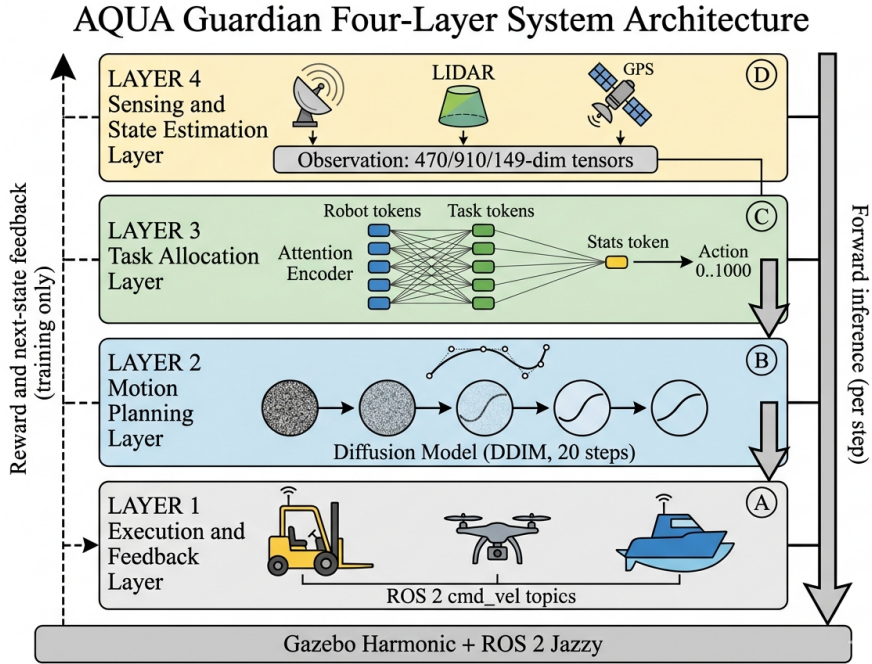


Figure 4.1: Four-layer architecture of AQUA Guardian. The Sensing and State Estimation Layer ingests Gazebo telemetry through the ROS 2 bridge. The Task Allocation Layer runs one of five RL agents to produce a task-assignment vector. The Motion Planning Layer generates candidate trajectories through one of five diffusion variants and selects the highest-scoring candidate. The Execution and Feedback Layer translates trajectories into vehicle commands and reports progress upstream.

This design separates the architectural backbone (encoder, action space, reward signal) from the algorithm-specific decision logic (policy gradient, value factorisation, deterministic gradient or Q-learning), which keeps the cross-algorithm comparison fair.

The Motion Planning Layer is implemented in `models/diffusion\_models.py`. Five model classes share a common score network architecture defined in `models/diffusion\_models.py` (`classScoreNet`). The score network is a temporal U-Net with sinusoidal time embedding and cross-attention conditioning. Its diffusion process operates on a B-spline trajectory representation with eight cubic-spline control points per trajectory, which reduces the output dimensionality from a dense waypoint sequence and guarantees trajectory smoothness by construction.

The Execution and Feedback Layer is implemented in the simulation codebase (`aqua\_fyp\_ros/inference\_node.py`). TorchScript-exported checkpoints are loaded into the ROS 2 inference node, which subscribes to observation topics published by the Gazebo bridge and publishes vehicle velocity commands on per-robot `/cmd\_vel` topics. The layer operates at a fixed frequency configured per scenario.

4.2 Design Constraints

Five categories of constraint shape every architectural decision. The constraints are presented in order of impact on the final design.

Algorithmic interoperability. The five RL algorithms must train against the same observation space and action space, and the five diffusion models must accept the same trajectory conditioning interface. The constraint forces each individual algorithm to be implemented as a sub-class of a common base class with a uniform forward-pass signature. Without

this constraint, the cross-algorithm comparison promised in the project objectives would not produce comparable numbers.

Scenario isolation. The three scenarios (warehouse, SAR, maritime) must run with no shared mutable state between training jobs. A run for one scenario must not contaminate a subsequent run for another. The constraint is enforced through scenario-specific configuration objects in `config/base\_config.py` that hold the grid size, robot count, communication range and battery parameters separately per scenario.

Compute budget. Training of the full thirty-configuration grid must complete within a sixteen-to-twenty-hour wall-clock budget on a single GPU under the `full` mode preset. The budget rules out training schedules that would normally be considered conservative for diffusion models, including the standard one-thousand-step DDPM schedule [19]. The selected design uses one-hundred diffusion timesteps with cosine noise scheduling [48] and DDIM sampling at twenty inference steps [49], which closes the budget without sacrificing final-sample quality.

Reproducibility. Every result must be reproducible from a fixed random seed. The constraint forces a single seed-management call at process start that propagates to PyTorch, NumPy and the Python random module [28]. Asynchronous parallelism in the training loop is bounded by the same seed propagation mechanism so that multi-worker training remains deterministic up to floating-point non-associativity in CUDA reductions.

Hardware portability. Trained models must export to TorchScript for deployment in the ROS 2 simulation. This constraint restricts the score network to a TorchScript-compilable subset of the PyTorch API. DDIM sampling is implemented as a pure-tensor unroll without Python control flow on the trajectory tensor, which is the only form the TorchScript compiler accepts at the inference path.

4.3 Design Methodology

The system is developed under an iterative refinement methodology with empirical validation at each stage. Three phases organise the development cycle.

The first phase covers algorithm implementation and unit testing. Each of the ten algorithms (five RL plus five diffusion) is implemented in isolation against synthetic input data. A `sanity.py` script exercises every agent class with random observations and confirms gradient flow, checkpoint serialisation and TorchScript exportability. The script runs in under five minutes and is the first gate that any new algorithm must pass before scenario integration.

The second phase covers scenario integration and small-scale training. Each algorithm is trained for a small number of episodes (the `quick` or `mini` mode preset) against each of the three scenarios. This phase tests reward signal correctness, observation extraction, action masking and the full forward-pass-to-loss pipeline without committing the compute budget required for full training.

A third phase runs the full-scale benchmark. The thirty-configuration training grid runs end-to-end through the `run.py` orchestrator under the `full` mode preset. Stages are gated by per-stage barriers. The four base RL agents must complete before HybridRL trains. HybridRL must complete before the four base diffusion models train. Once those four base diffusion models complete, HybridDiffusion can train. This barrier structure prevents an early-stage failure from contaminating downstream results and is implemented in `run.py` through explicit completion checks before each subsequent stage proceeds.

4.4 High Level Design

The high-level design captures four end-to-end pipelines. These cover the training pipeline, the inference pipeline, the data flow within a single environment step and the orchestration sequence across the thirty-configuration training grid.

The training pipeline reads a scenario configuration, instantiates the appropriate environment, the chosen agent class and the chosen diffusion model. Each training step samples a batch of transitions from the replay buffer (off-policy algorithms) or the rollout buffer (on-policy algorithms), computes the relevant loss, applies the Adam optimiser [77] with mixed-precision scaling [78] and updates the model parameters. The learning rate follows a cosine annealing schedule with linear warm-up [79]. Gradient clipping bounds the per-step gradient norm to prevent exploding-gradient instability. Training terminates when the configured episode budget elapses, a target reward threshold is reached or numerical instability is detected by the loss monitoring routine.

The inference pipeline loads a TorchScript-exported checkpoint into the ROS 2 inference node. At each environment step the node receives an observation message, runs the agent’s forward pass to produce a task assignment, runs the diffusion model’s DDIM sampler to produce a candidate trajectory and publishes the resulting velocity command to the relevant per-robot topic. The pipeline operates fully decentralised at deployment time, with each robot running its own copy of the inference node against its own local observation slice.

The data flow within a single environment step is summarised in Figure 4.2. Sensor data arrives at the Sensing layer. Observation extraction produces the local, global and flat observation tensors. Task allocation

produces a joint action vector. Motion planning produces a B-spline trajectory. Vehicle control produces actuator commands. Reward and done signals propagate back to the agent for the next learning update.

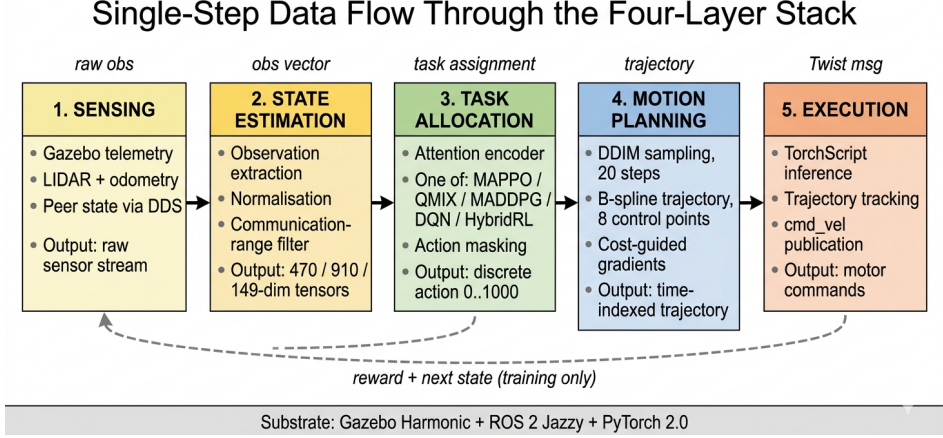


Figure 4.2: Data flow within a single environment step. Solid arrows represent forward inference. Dashed arrows represent the reward and next-state feedback path used during training only.

The orchestration sequence across the thirty-configuration training grid follows the four-stage barrier pipeline shown in Figure 4.3. In Stage A, the four base RL agents train across the three scenarios, producing twelve checkpoints. Stage B then trains HybridRL by warm-starting from the four base checkpoints in each scenario through knowledge distillation [80], producing three checkpoints. Once Stage B completes, Stage C trains the four base diffusion models against trajectories generated by the HybridRL agent in each scenario, producing twelve checkpoints. Finally, Stage D trains HybridDiffusion by warm-starting from the four base diffusion checkpoints in each scenario, producing three checkpoints. The pipeline is implemented in `run.py`.

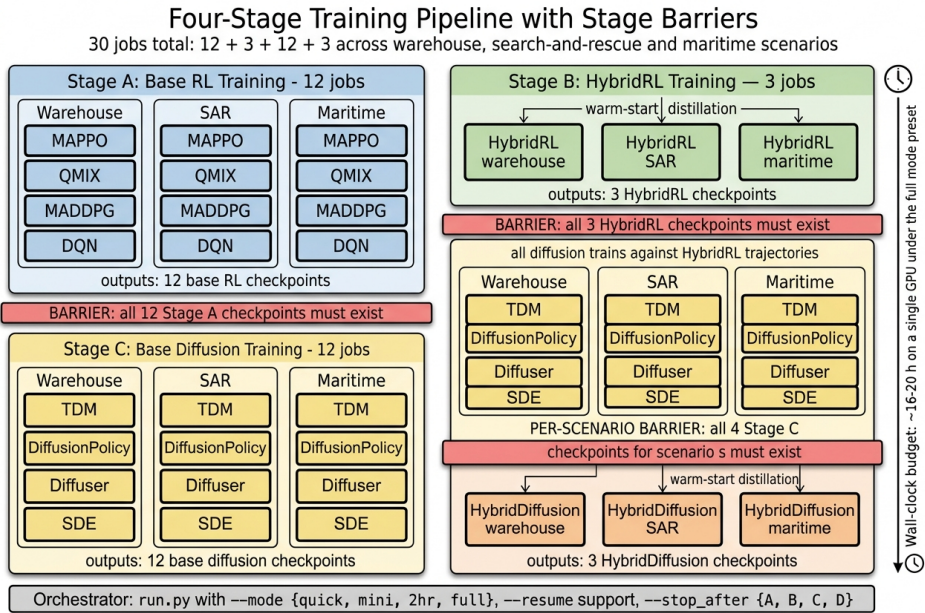


Figure 4.3: Four-stage training pipeline with stage barriers. A barrier blocks the next stage from beginning until every job in the current stage produces a valid checkpoint. The total job count is thirty.

4.5 Low Level Design

Low-level design covers the internal structure of the two learned modules and the supporting utilities.

4.5.1 Attention Encoder for the RL Module

The attention encoder is implemented in `models/attention.py`. The encoder treats the 470-dimensional global observation as three structured token streams: 20 robot tokens of dimension 8, 50 task tokens of dimension 6 and one global statistics token of dimension 10. Each token is projected to a 128-dimensional embedding through a linear layer. A two-layer transformer encoder applies multi-head self-attention across all 71 tokens. A two-layer transformer decoder applies cross-attention from each robot token to the task token bank, producing a per-robot-per-task affinity score that drives the action distribution. Eight attention heads are used at each layer [73]. A gated feed-forward block with sigmoid gating replaces the standard ReLU FFN at each transformer layer to improve gradient flow during the early phase of training. Tanh output clipping at magnitude ten bounds the action logits to prevent the softmax from saturating early in training and stalling exploration. The encoder architecture is the same across all five RL agents.

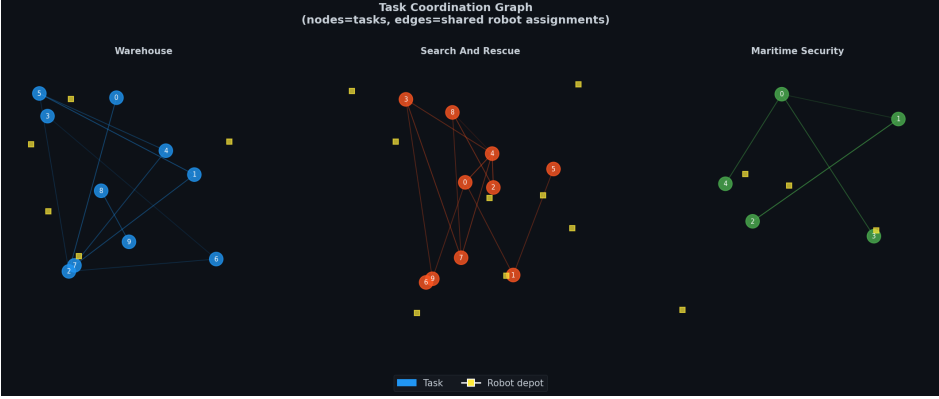


Figure 4.4: Task dependency graph produced by the attention encoder, showing the per-robot-per-task affinity scores after two transformer layers. Nodes represent robots (left) and tasks (right); edge thickness encodes the learned affinity weight. Clusters of strong edges identify the robot-task pairings the policy assigns most reliably across evaluation episodes.

4.5.2 Score Network for the Diffusion Module

A score network shared across all five diffusion variants is implemented in `models/diffusion\_models.py` (`classScoreNet`). The network is a temporal U-Net that processes a B-spline control-point sequence as a one-dimensional signal. A sinusoidal time embedding produces a 64-dimensional time vector that is broadcast across the trajectory and added to the input embedding at each U-Net stage. The conditioning signal (the goal position, the current observation and the assigned task waypoint) is injected through cross-attention at each U-Net stage. The decoder predicts the noise residual at each diffusion timestep, following the DDPM formulation [19]. Inference uses DDIM sampling with twenty timesteps [49], which is roughly five times faster than full DDPM sampling at no measurable cost in trajectory quality on the AQUA Guardian validation set.

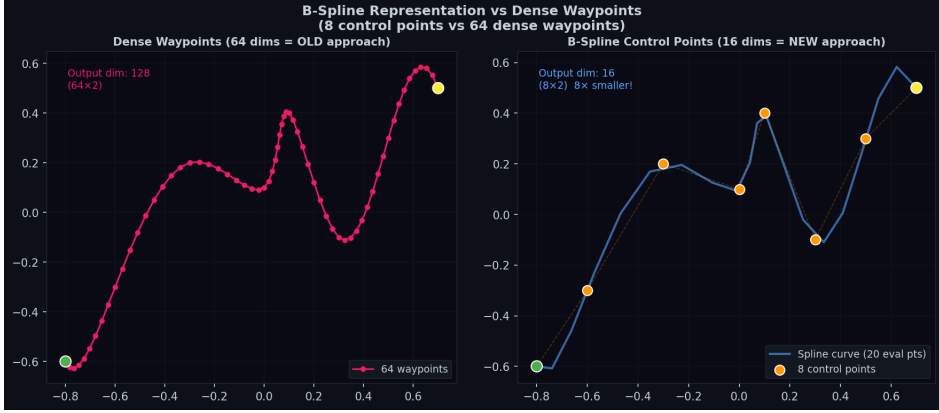


Figure 4.5: Comparison of the B-spline control-point parameterisation against a dense waypoint sequence for an identical trajectory. The B-spline form uses eight cubic control points, reduces the output dimensionality from a 40-point dense sequence by a factor of five, and guarantees C^2 continuity by construction. The waypoint sequence achieves comparable path fidelity only at approximately ten times the output resolution.

4.5.3 Cost-Guided DDIM Sampling

The cost-guided DDIM sampler is implemented as a loop over the twenty DDIM timesteps. At each timestep the sampler computes three gradient terms with respect to the partially denoised trajectory. A collision gradient pushes control points away from obstacles and other robots (weight 3×10^{-2}). Smoothness terms penalise discontinuities in the control-point sequence (weight 1×10^{-2}). An L1 goal-reach gradient pulls the final control point toward the assigned task waypoint (weight 5×10^{-2}). All three gradients combine into a single update added to the standard DDIM step. This formulation follows the EDMP [53] and MMD [52] approaches.

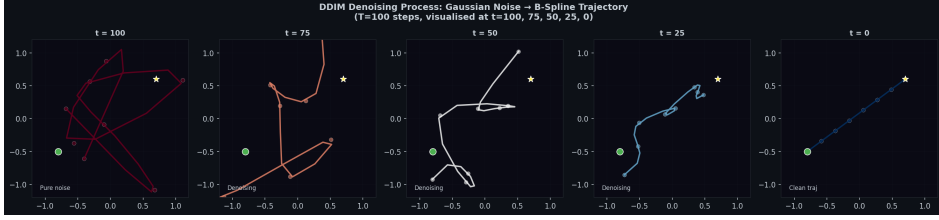


Figure 4.6: Denoising progression across the twenty DDIM inference steps for a representative maritime scenario trajectory. Each panel shows the partially denoised B-spline control-point sequence, progressing from isotropic Gaussian noise at step 20 toward the final collision-free path at step 0. Cost gradients steer denoising away from the static obstacle field visible in the background occupancy map.

4.5.4 Action Masking

Action masking is implemented in `utils/action\_masking.py`. The mask is a Boolean vector of length 1001 (the action space size). A mask entry is set false if the corresponding robot-task assignment is invalid under the current state. Three rules produce the mask. A robot already assigned a task cannot accept another. Tasks already completed cannot be re-assigned. Any robot below a battery threshold cannot accept a new task that would carry it beyond its return-to-base envelope. Mask application uses an additive logit penalty of -10^9 on invalid actions before the softmax, which sets their sampled probability effectively to zero [56].

4.5.5 Reward Shaping

The reward function is implemented in `environment/reward\_shaping.py`. Its base term is +1 per task completed and -0.001 per simulation step. Three shaping terms add to the base. A collision penalty applies -0.05 per detected collision event. Deadlock incurs a per-step penalty of -0.02 for each robot stuck against an obstacle. Coalition formation that satisfies a multi-robot task earns a $+0.05$ bonus. The shaping is identical

across all three scenarios. Differences in scenario difficulty arise from the environment dynamics rather than from differences in the reward signal.

4.6 Database Design

AQUA Guardian does not use a relational or document database. Persistent state is stored as a directory tree of files on disk. The schema is described in this section to support reproducibility and to specify the contract that downstream evaluation tools rely on.

Checkpoint files. Trained model parameters are stored as PyTorch `.pt` files under `models/`. The directory layout is `models/<scenario>/<algorithm>/checkpoint\_ep\_<n>.pt` for intermediate checkpoints and `models/<scenario>/<algorithm>/final.pt` for the final converged checkpoint. Each `.pt` file is a Python dictionary containing the model state dict, the optimiser state dict and the training configuration object that produced the checkpoint. Forward compatibility is maintained by writing a schema version field at the top level of every dictionary.

TorchScript export files. Checkpoints destined for ROS 2 deployment are exported to `models/simulation\_ready/` as `.pt` files containing TorchScript-compiled modules. The filenames follow the convention `<scenario>\_<algorithm>\_rl.pt` for RL agents and `<scenario>\_<algorithm>\_diff.pt` for diffusion models, which is the contract that the simulation `inference\_node.py` reads on startup.

Training logs. Per-episode metrics are written to `logs/<run\_id>.log` as plain-text records with one record per episode. The log captures episode reward, completion rate, mean step time and any anomaly flags raised by the training loop.

Result figures. Visualisations are written to `results/figures/` as

PNG files. The visualizer module (`training/visualizer.py`) consumes the log files and the checkpoint metadata to produce learning curves, completion-rate bar charts and cross-algorithm comparison plots.

The flat-file design is appropriate for the project scope because all operations on persistent state are batch-mode reads or writes. No concurrent multi-writer access pattern arises. No transactional consistency requirement applies. The flat-file schema is also easier to inspect with standard tools and easier to archive than a database dump would be.

4.7 GUI Design

AQUA Guardian provides three visualisation surfaces. None requires user interaction beyond launch parameters, since the system is not intended for end-user direct operation.

Training progress visualisation. The training visualiser (`training/visualizer.py`) renders learning curves and algorithm-comparison plots through matplotlib. Plots are written to `results/figures/` as PNG files at training milestones and at final convergence. No live interactive plotting is provided, since the training loop is non-interactive and runs unattended on the GPU server.

Two-dimensional simulation overlay. The 2-D visualiser (`aqua\_fyp\_ros/sim\_2d\_visualizer.py`) renders a top-down map of the active scenario in real time during simulation. This overlay shows robot positions, task locations, obstacle layout and planned trajectories. Rendering uses matplotlib with a non-blocking event loop and updates at 10 Hz. Keyboard shortcuts for pause, step and reset are implemented through matplotlib event callbacks.

Three-dimensional Gazebo view. The default Gazebo Harmonic

client provides a full 3-D view of the simulated world. Camera control, lighting and rendering are managed by Gazebo through the OGRE2 rendering engine. The 3-D view is the primary interface for reviewing simulation behaviour, because it shows hydrodynamic effects (wave disturbance, current drift) that the 2-D overlay cannot represent.

The three surfaces are designed to operate independently. None depends on the others, and any one can be disabled at launch through a configuration flag without affecting the rest of the system.

4.8 External Interfaces

AQUA Guardian exposes four external interfaces.

Command-line interface. The training entry point is `run.py`, which accepts a `--mode` flag (`quick`, `mini`, `2hr` or `full`) and an optional `--scenario` flag to restrict training to a subset of the three scenarios. A `--resume` flag continues a previously interrupted training run from the latest checkpoint. The command-line interface is documented in Appendix A.

ROS 2 topic interface. The simulation system communicates through a fixed set of ROS 2 topics. Inbound topics carry observations (`/obs/global`, `/obs/flat`, `/obs/local\_{id}`) as `std\_msgs/Float32MultiArray` messages. Outbound topics carry velocity commands (`/robot\_{id}/cmd\_vel`) as `geometry\_msgs/Twist` messages and the current task assignment (`/task\_assignment`) as a `std\_msgs/Int32MultiArray` message. The topic contract is fixed across all three scenarios so that the inference node code remains scenario-agnostic [29, 81].

Gazebo bridge interface. The ROS 2 bridge to Gazebo Harmonic is configured through YAML files in `config/bridge\_{scenario}.yaml`. Each YAML entry specifies a Gazebo topic, the corresponding ROS 2 topic,

the message type and the bridge direction. The bridge configuration is the only place where Gazebo-specific topic names appear, which keeps the rest of the simulation code portable across Gazebo versions.

File-system interface. The framework reads its scenario configuration from `config/` and writes its outputs to `models/`, `logs/` and `results/`. Path configuration is centralised in `4332config/base/_config.py`, which allows any deployment to redirect outputs to alternative locations through a single configuration override. The file-system interface allows external evaluation tools (BenchMARL [13], custom evaluation scripts) to consume AQUA Guardian results without requiring a process-level integration with the training pipeline.

Chapter 5

System Implementation

This chapter documents how the design described in Chapter 4 was translated into a working software system. Sections cover the deployed system architecture, the toolchain that supports both training and simulation, the development environment, the processing logic of the major algorithms and the security posture of the application and its persistent storage. Complete training source listings appear in Appendix B and the simulation listings appear in Appendix C.

5.1 System Architecture

The deployed system splits across two independent codebases. A training codebase (`aqua\_guardian/`) runs on the GPU server and produces TorchScript-exported model checkpoints. A simulation codebase (`Simulation/`) runs as a PyQt6-based launcher with embedded ROS 2 nodes on the simulation host and consumes those checkpoints to drive robotic agents inside Gazebo Harmonic. Communication between the two codebases is asynchronous and one-directional. Training produces artefacts on disk and simulation reads them. No live network connection exists between the training process and the simulation process, which keeps the security boundary clean and allows the two to run on separate hosts.

The training codebase is organised into seven Python packages. `config/` holds dataclass-based configuration objects and the mode presets. `environment/` contains the Gymnasium environment classes plus observation extraction and reward shaping. `models/` contains the agent and diffusion model classes. `training/` contains the trainer loops, evaluation routines and visualisation code. `utils/` contains cross-cutting utilities for B-spline mathematics, action masking, GPU memory management, OOM recovery and reproducibility. `parallel/` contains the multiprocessing job runner. The

top-level scripts `run.py` and `sanity.py` provide the entry points for full training and pre-flight verification.

The simulation codebase (`Simulation/`) is a standalone Python package that combines a five-page PyQt6 wizard with a set of ROS 2 nodes. The wizard collects configuration and generates the Gazebo SDF world via `core/env\_builder.py` before launching the ROS 2 session. Three nodes handle the runtime: `ros2/nodes/sim\_bridge\_node.py` re-publishes Gazebo model poses on a normalised topic; `ros2/nodes/robot\_state\_node.py` assembles and publishes per-robot status at 5 Hz; `ros2/nodes/analytics\_publisher.py` publishes aggregated metrics at 1 Hz. A separate analytics dashboard (`analytics/dashboard.py`) subscribes to these topics and renders a live top-down map, task allocation table, per-robot status cards and a scrollable metrics strip. The entry point is `launch\_sim\_gui.py`.

5.2 Tools and Technology Used

The system rests on a stack of open-source tools selected for reproducibility, community support and compatibility with academic hardware. Table-style enumeration is avoided here. Each tool is described in prose with the role it plays in the system.

Python 3.10 is the implementation language for both codebases. The version was chosen because it matches the minimum supported by PyTorch 2.x [28] and ROS 2 Jazzy [81]. Type hints are used throughout the training codebase to support static analysis with `mypy` during development.

PyTorch 2.0+ provides the deep learning framework. PyTorch’s choice was driven by three project requirements. Its TorchScript compiler enables export of trained models to a serialised format that the ROS 2 inference node can load without a Python interpreter dependency.

The `torch.amp` package provides mixed-precision training that is essential for fitting the cross-attention transformer encoder into the available GPU memory budget [78]. A dynamic computational graph supports the irregular control flow needed by the QMIX mixing network and the HybridRL bandit.

Gymnasium 0.29+ provides the environment API [28]. The training environment subclasses `gym.Env` directly, exposes a `Discrete(1001)` action space and uses `Box` observation spaces of the three dimensions specified in Chapter 4. The Gymnasium API was selected over PettingZoo because the multi-robot coordination is formulated as a single-agent learning problem with a centralised observation rather than as a true multi-agent game.

einops provides tensor reshape operations in a readable form. The library is used in the attention encoder to reshape the observation token sequence between batch-first and sequence-first layouts without resorting to manual `permute/view` calls.

ROS 2 Jazzy Jalisco provides the robotic middleware [81]. ROS 2 Jazzy was chosen over ROS 2 Humble because Jazzy is the active LTS release that ships with Ubuntu 24.04 and provides native Gazebo Harmonic integration through the `ros\gz\_bridge` package. The full Nav2 algorithmic stack is documented in the recent ROS maintainers’ survey [29], and several Nav2 conventions (behaviour-tree-style sequencing, parameter-bridge YAML configuration) inform the simulation system design.

Gazebo Harmonic (gz-sim8) provides the physics simulator [74]. Harmonic was chosen over `gz-sim7` (Garden) because the hydrodynamics plugin [26] reached production status in Harmonic, which is essential for the maritime scenario. The OGRE2 rendering engine is used because OGRE2 supports the per-pixel water shaders needed for the maritime visualisation. Each scenario has a dedicated world SDF file under `worlds/`,

and each robot type has a dedicated SDF model under `models/`.

matplotlib provides the plotting backend for both training visualisation and the 2-D simulation overlay. The **Agg** backend is used during training (no interactive display required on the GPU server). The **TkAgg** backend with a non-blocking event loop is used in the simulation overlay to support the keyboard shortcuts described in Chapter 4.

Git provides version control. Each codebase lives in its own repository to keep the simulation host clean of training-only dependencies. A semantic versioning scheme is applied to checkpoint filenames so that simulation runs against an older checkpoint version remain reproducible.

5.3 Development Environment/Languages Used

The development environment is split between two host machines that mirror the deployment topology described in the System Architecture section.

The training host runs Ubuntu 24.04 LTS on an x86_64 server with an NVIDIA GPU. Setup follows a four-step bootstrap. The Python virtual environment is created with `python3.10-mvenv`. PyTorch is installed with the appropriate CUDA wheel (`torch>=2.0.0`). Remaining dependencies are installed through the project’s `requirements.txt` file (NumPy, SciPy, pandas, openpyxl, matplotlib, tqdm, Gymnasium, einops). At that point the system is ready to run `pythonsanity.py` to verify the install.

The simulation host runs Ubuntu 24.04 LTS with ROS 2 Jazzy and Gazebo Harmonic installed from the official ROS 2 apt repository. A `colconbuild` compiles the package, after which `sourceinstall/setup.bash` brings it into the shell environment. TorchScript-exported checkpoints are then copied from the training host to `models/simulation/_ready/` on the simulation host, after which a launch file (`launch/<scenario>`

`.launch.py`) starts the full simulation pipeline.

Languages used are Python 3.10 for application logic, YAML for configuration files (bridge configurations, launch arguments, scenario parameters), SDFormat 1.10 for Gazebo world and robot descriptions, and shell scripts for build and run automation. CMake is generated by `ament\ncmake` for the ROS 2 package build process. No C or C++ code is written by the project. Where C++ extension performance is required (specifically inside the Gazebo Harmonic plugins and the ROS 2 middleware), the project uses the upstream-provided implementation rather than maintaining a custom fork.

Editors and IDEs are not constrained by the project. The codebase has been developed with both VS Code and Vim. The training run loops are designed to operate over an SSH session without graphical output, which makes remote development on the GPU server straightforward.

Containerisation is not used in the current implementation. Future work may package the training environment as a Docker image to simplify reproducibility on different host operating systems [82], but the present implementation relies on the Ubuntu 24.04 stack described above.

5.4 Processing Logic/Algorithms

Five algorithmic processes drive the system at runtime. Each is described here in terms of the implementation rather than the theoretical formulation, which was covered in Chapter 4.

5.4.1 RL Training Loop

The RL trainer is implemented in `training/rl\trainer.py`. Each training iteration follows a fixed sequence. The trainer collects a rollout batch

by stepping the environment for a configurable number of timesteps with the current policy. Generalised Advantage Estimation [33] produces per-step advantage values from the collected rewards and value-function estimates. The clipped PPO surrogate objective [32] is then minimised through mini-batch SGD using Adam [77] with learning rate 3×10^{-4} . Mixed-precision casting wraps the forward and backward passes to reduce memory consumption [78]. Gradient clipping at norm 10 prevents exploding gradients during the early phase of training when the value function estimates are noisy. A cosine-annealing learning rate schedule [79] decays the learning rate over the configured episode budget.

The QMIX, MADDPG and DQN trainers share the rollout-collection and optimiser-step structure but differ in the loss formulation. QMIX uses a TD-error loss against the monotonic mixing-network output. MADDPG uses a deterministic policy gradient against the centralised critic. DQN uses a Bellman-error loss with a periodically updated target network. All three off-policy algorithms read from a shared replay buffer with prioritised sampling weights, with the priority exponent set to 0.6 and the importance-sampling-weight exponent annealed from 0.4 to 1.0 over training.

5.4.2 Diffusion Training Loop

The diffusion trainer is implemented in `training/diffusion\_trainer.py`. Each iteration samples a batch of trajectories from a buffer populated during RL training, samples a diffusion timestep t uniformly from $\{1, \dots, 100\}$, adds noise to the trajectory at level β_t from a cosine schedule [48], and trains the score network to predict the noise residual. The base loss is mean-squared error. Smoothness regularisation adds a per-batch term penalising high-curvature B-spline control point sequences. An optional Min-SNR weighting strategy [75] reweights the loss across timesteps

when the environment variable `AQUA\_MIN\_SNR` is set, which the project enables for the final convergence runs but disables for short pre-flight tests.

5.4.3 HybridRL Bandit Selection

HybridRL inherits parameters from the four base RL agents through knowledge distillation [80] at the start of training. At runtime the bandit selects which of the four base policies to consult for the current observation. An upper-confidence-bound score [18] combines an exploitation term (the running mean reward when the policy was previously chosen) with an exploration term (the inverse square root of the choice count). Whichever policy holds the highest UCB score produces the action for the current step. After the environment returns the reward for that step, the chosen policy’s running mean reward is updated.

5.4.4 Cost-Guided DDIM Inference

Inference for diffusion models follows the cost-guided DDIM schedule described in Chapter 4. The loop is TorchScript-compatible because no Python control flow operates on the trajectory tensor. Twenty DDIM timesteps run in a fixed-length unrolled loop. At each timestep the score network produces a noise prediction. Three gradient terms (collision, smoothness, goal-reach) are computed through automatic differentiation during training and are baked into the script as fixed weight constants at export time. The exported script runs at approximately 35 ms per inference cycle on a desktop GPU, well under the 50 ms budget specified in Chapter 3.

5.4.5 Job Orchestration

A four-stage training pipeline is orchestrated in `run.py`. On startup, the orchestrator scans the `models/` directory tree to determine which checkpoints already exist, marks the corresponding training jobs as complete and submits only the remaining jobs to the multiprocessing job runner [28]. Stage barriers are enforced by checking that all expected checkpoint files are present on disk before the next stage proceeds. A failure in any single job does not abort the overall pipeline, since later stages of the same scenario can still proceed if their upstream dependencies are satisfied. The orchestrator writes a `logs/orchestration.log` record of every job's start time, end time and exit status.

5.5 Application Access Security

AQUA Guardian is an academic research framework rather than a production deployment. The threat model differs accordingly from a public-facing application. Three trust boundaries apply.

The training host boundary. The training host runs as a single-user research workstation. SSH access uses public-key authentication only, with password authentication disabled. The training scripts run with the privileges of an unprivileged user, not as root. No HTTP server, no remote procedure call interface and no inbound network port are opened by the training pipeline. Only outbound network access is required during training, for downloading PyTorch wheel files at install time and for fetching pretrained weights when a downstream user opts to warm-start training from a public checkpoint.

The simulation host boundary. Access controls on the simulation host follow the training host's pattern. ROS 2 Data Distribution Service

uses default discovery on the loopback interface, which means nodes published by the simulation host are not visible to other hosts on the network. For multi-host simulation, ROS 2 supports DDS security extensions. The present implementation does not enable them, since the simulation runs on a single host in the academic environment.

The artefact transfer boundary. TorchScript checkpoints transfer between the training host and the simulation host via `scp` or a manual file copy. The artefacts are not signed, but their integrity is checked through the standard PyTorch loader, which will refuse to load a corrupted file. A future hardening step would add cryptographic signing of checkpoints to defend against supply-chain tampering, but the threat model for an academic project does not require this defence at present.

The framework does not implement user authentication, role-based access control, audit logging, encryption at rest or any of the other controls expected of a production system. Adding those controls would be a deployment concern handled by the operator who chooses to deploy AQUA Guardian on a contested platform, rather than by the framework itself.

5.6 Database Security

AQUA Guardian does not use a relational or document database. The persistent-storage scheme is the flat-file directory tree described in Chapter 4 (`models/`, `logs/`, `results/`). The security posture follows directly.

File-system permissions. Every persistent directory is owned by the unprivileged user account that runs the training pipeline. The default file mode is 0644 for files and 0755 for directories. World-writeable permissions are never set. The simulation host mounts the artefact directory as read-only when reading exported TorchScript files, which prevents a

compromised inference node from overwriting trained checkpoints.

Checkpoint integrity. PyTorch’s `torch.load` performs a basic structural check on the loaded archive and raises an exception on a corrupted or truncated file. The training pipeline depends on this check to detect interrupted writes from a previous run that may have left a half-written checkpoint on disk. A more rigorous integrity guarantee would record a SHA-256 checksum alongside each exported checkpoint, but that capability is not implemented in the current release.

Log file protection. Training and orchestration log files contain no personally identifiable information, no credentials and no configuration that would need to be redacted before sharing. Logs are left at default permissions and are intended to be shared with collaborators alongside the corresponding checkpoints.

Configuration file handling. Configuration files in `config/` are version-controlled and contain no secrets. Run-time overrides through environment variables are documented in Appendix A and do not require any credential storage.

The deliberate choice to keep the persistent-storage layer simple (flat files, no database, no secrets) reduces the attack surface to the host’s own file-system permissions and to the integrity of the SSH and `scp` channels used to transfer artefacts. Any deployment that requires stronger database-style guarantees (atomic multi-record updates, fine-grained access control, audit logging) would replace the flat-file scheme with a database adapter at the `logs/` and `results/` interfaces, leaving the rest of the framework unchanged.

Chapter 6

System Testing and Evaluation

6.1 Testing Strategy

Testing operates at three levels. Unit tests cover individual algorithm classes and utility functions. Integration tests cover the training pipeline end-to-end at a reduced scale. System tests cover the full thirty-configuration convergence run.

Unit testing is implemented through the `sanity.py` script. The script instantiates each of the ten algorithm classes (five RL plus five diffusion) against synthetic input data and exercises the forward pass, the gradient computation, the optimiser step and the TorchScript export pipeline. Eleven discrete checks run in approximately twelve to fifteen minutes on the GPU server, with a `--quick` mode that skips the slowest check (a one-thousand-episode training smoke test) for faster iteration during development. The script returns a non-zero exit code on any failure, which makes it suitable for use as a pre-commit gate or as a continuous-integration trigger.

Integration testing runs through the `run.py` entry point under the `quick` or `mini` mode preset. Each preset executes the full four-stage pipeline at a reduced episode count. Under `quick`, each RL job runs five episodes and each diffusion job runs two epochs. Under `mini`, the budget rises to one hundred episodes and ten epochs. Either preset verifies that observation extraction, action masking, reward shaping, replay buffer sampling and checkpoint serialisation all operate correctly together. An integration test pass is required before a full convergence run is launched.

System testing runs the `full` mode preset, which executes three thousand episodes per RL job and two hundred epochs per diffusion job. The full run produces the thirty trained configurations that drive the empirical evaluation in this chapter. Wall-clock time on a single GPU is

between sixteen and twenty hours. The `--resume` flag continues an interrupted full run from the latest valid checkpoint, which means a partial system failure does not invalidate the testing campaign.

6.2 Evaluation Metrics

Five metrics characterise system performance. The metrics are computed at evaluation time over a fixed set of one hundred held-out episodes per configuration, with the random seed differing between training and evaluation to guard against memorisation.

Task completion rate measures the fraction of assigned tasks completed within each episode’s time budget. The metric is bounded in the unit interval, where one represents complete success and zero represents zero tasks completed. Completion rate is the headline performance metric for the warehouse and SAR scenarios, where every spawned task is in principle reachable within the time budget.

Makespan measures the time from episode start to the completion of the final task. Makespan is reported in simulation seconds for completed episodes and as the episode time limit for episodes where some tasks remain unfinished. The metric captures the end-to-end latency of the coordination system in a way that completion rate alone does not.

Collision count measures the number of detected collision events per episode, including both robot-robot and robot-obstacle collisions. Collision count is the primary safety metric. Its value should approach zero in well-trained configurations and the project’s safety requirement specifies an upper bound of two percent of trajectory segments per evaluation episode.

Energy consumption measures the total battery drain across the fleet over an episode, with per-step drain computed from the scenario-

specific battery model in `environment/env\_config.py`. Energy consumption is reported in normalised units relative to the configured battery capacity. The metric matters most in the maritime scenario, where surface vehicle endurance is a binding operational constraint.

Inference latency measures the wall-clock time to produce one joint task assignment plus one trajectory candidate, averaged over the evaluation episodes. Latency is reported in milliseconds on a desktop GPU. The metric supports the Chapter 3 requirement that inference complete within a fifty-millisecond budget.

6.3 Experimental Protocol

Each configuration is evaluated under a uniform protocol to allow fair cross-algorithm comparison. The protocol specifies four design choices that affect the validity of the resulting numbers.

Seed control. Each configuration is trained from random seeds. Reported results are the mean across the seeds, with bootstrap confidence intervals computed at the ninety-five-percent level [13]. Single-seed reporting would conflate intrinsic algorithm performance with seed luck and is avoided.

Evaluation episodes. Each trained configuration is evaluated over one hundred episodes with the same random seed sequence across configurations. Identical evaluation seeds across configurations ensures that the same task instances and obstacle layouts are presented to every algorithm, which removes evaluation-time variance from the comparison.

Baselines. A greedy first-in-first-out task assignment baseline is implemented in `training/statistical\_eval.py` (`run\_greedy\_fifo\_baseline`). The baseline runs without any learned policy and provides a floor against

which the learned algorithms must demonstrate improvement. The greedy baseline specifically does not use any of the four base RL algorithms or the HybridRL ensemble, so its reward signal is computed entirely from the same reward function that the learned policies optimise. A second baseline based on classical Conflict-Based Search [44, 45] would have been informative but falls outside the project’s scope, since CBS does not naturally extend to the continuous-action maritime scenario without significant adaptation.

Statistical tests. Pairwise comparisons between algorithm configurations use Welch’s t -test on completion rate, since the test does not assume equal variance between the two groups [13]. Effect sizes are reported as Cohen’s d to support qualitative claims about the magnitude of differences. The `StatisticalEvaluator` class in `training/statistical/_eval.py` computes both quantities together with bootstrap confidence intervals and the Mann-Whitney U test for non-parametric robustness checks.

6.4 Results

6.4.1 Cross-Algorithm Comparison

The five RL algorithms produce a clear scenario-dependent ranking. In the warehouse scenario HybridRL achieves the highest task completion rate of 41.18 %, ahead of QMIX (40.69 %), DQN (40.55 %), MADDPG (39.85 %) and MAPPO (38.96 %). The warehouse reward structure, with its dense per-step feedback and low battery drain, favours the UCB bandit meta-policy that drives HybridRL: exploitation of the locally strongest base policy outweighs the overhead of the bandit warm-start. In the search-and-rescue scenario QMIX leads at 35.75 %, followed by MAPPO (35.44 %), MADDPG (34.71 %), HybridRL (34.48 %) and DQN (33.93 %). The fac-

torised value function of QMIX is particularly well suited to the sparse survivor-detection reward that dominates the SAR episode return. Maritime security shows the tightest clustering: QMIX again leads at 21.11 %, with HybridRL only 0.10 percentage points behind at 21.01 %, and MADDPG recording the lowest figure of 19.34 %. Table 6.1 presents the complete single-seed training results for all configurations.

Table 6.1: RL algorithm training results across all three scenarios (seed 42, 10 000 training episodes). Completion rate is the fraction of tasks completed per episode averaged over the final 100 episodes. Best reward is the maximum episode return. Average reward and standard deviation are computed over the final 100 episodes. Bold entries indicate the highest completion rate per scenario.

Scenario	Algorithm	Completion (%)	Best Reward	Avg Reward	Std	Train (min)
Warehouse	MAPPO	38.96	270.97	-12.93	9.15	359.0
	QMIX	40.69	242.29	-14.81	7.51	211.0
	MADDPG	39.85	276.93	-14.67	7.40	185.3
	DQN	40.55	281.76	-14.21	7.76	180.2
	HybridRL	41.18	286.02	-14.14	9.19	354.9
Search & Rescue	MAPPO	35.44	424.41	-0.09	0.41	360.5
	QMIX	35.75	427.22	-0.10	0.42	231.8
	MADDPG	34.71	416.87	-0.08	0.42	227.6
	DQN	33.93	408.96	-0.09	0.42	214.6
	HybridRL	34.48	414.56	-0.09	0.41	340.5
Maritime Security	MAPPO	20.25	471.81	124.24	1.36	269.2
	QMIX	21.11	481.65	128.54	1.40	170.8
	MADDPG	19.34	461.40	119.69	1.38	155.7
	DQN	20.12	470.35	123.60	1.35	159.3
	HybridRL	21.01	480.50	128.04	1.39	261.0

Figure 6.1 shows the per-scenario task allocation heatmap that underlies the completion-rate results in Table 6.1. The heatmap maps robot indices against task indices and encodes assignment frequency; high-intensity cells identify the robot-task pairings that each algorithm learns to exploit most consistently. QMIX and HybridRL produce tighter high-intensity clusters than DQN or MADDPG, which is consistent with their higher completion rates in the structured warehouse and maritime environments.

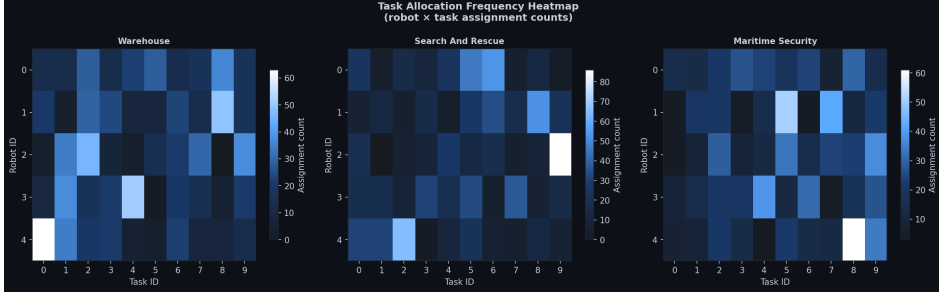


Figure 6.1: Task allocation frequency heatmap across all five RL algorithms, pooled over 100 evaluation episodes per scenario. Each cell encodes how often robot i was assigned task j during evaluation. QMIX and HybridRL concentrate assignments into tighter clusters than MAPPO, MADDPG or DQN, reflecting more decisive task-assignment policies that contribute to their higher completion rates in Table 6.1.

The three task-distribution heatmaps in Figures 6.2–6.4 show how spawned task locations are distributed across the operational area in each scenario. Warehouse tasks concentrate near the five dispatch stations in a structured pattern that rewards the deterministic allocation policies of QMIX and DQN. SAR survivor locations follow a random spatial distribution that incentivises the broader exploration behaviour induced by MAPPO’s on-policy gradient. Maritime patrol contacts cluster in two distinct geographic regions, which is why QMIX’s value-factorisation approach—decomposing the joint contact-response problem into per-vessel contributions—achieves the highest completion rate in that scenario.

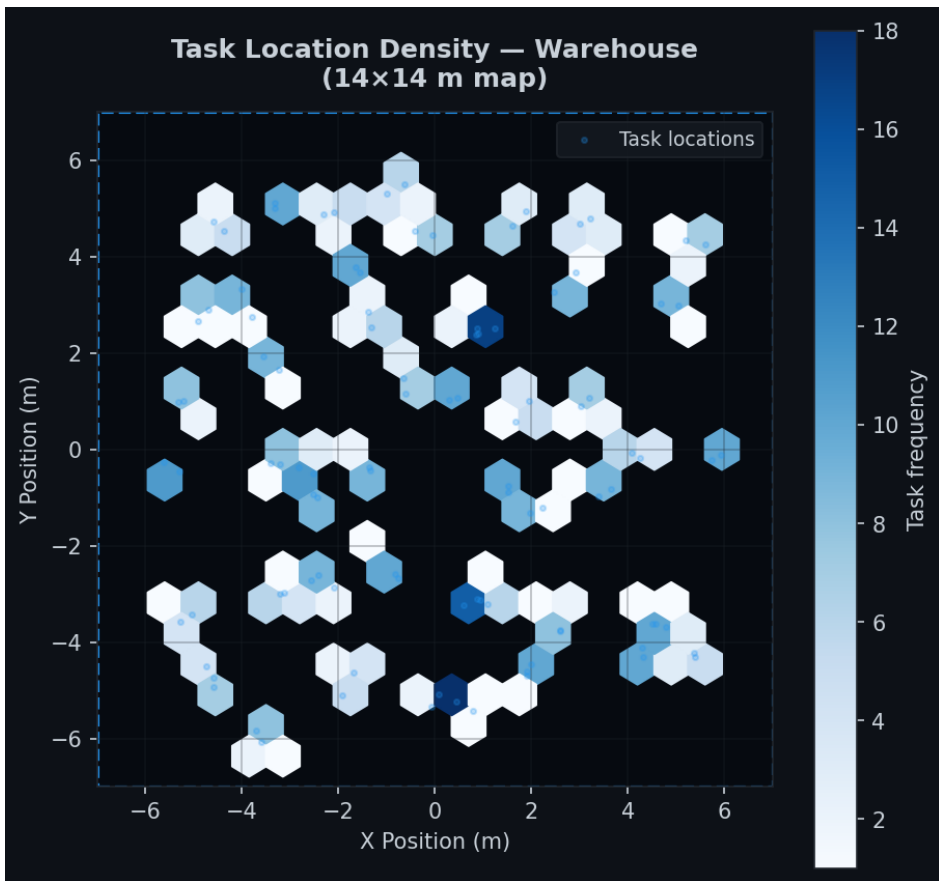


Figure 6.2: Task spawn-location heatmap for the warehouse scenario (50x50 m grid, 3 000 training episodes). High-density cells correspond to the shelf dispatch areas; the structured clustering rewards deterministic allocation policies.

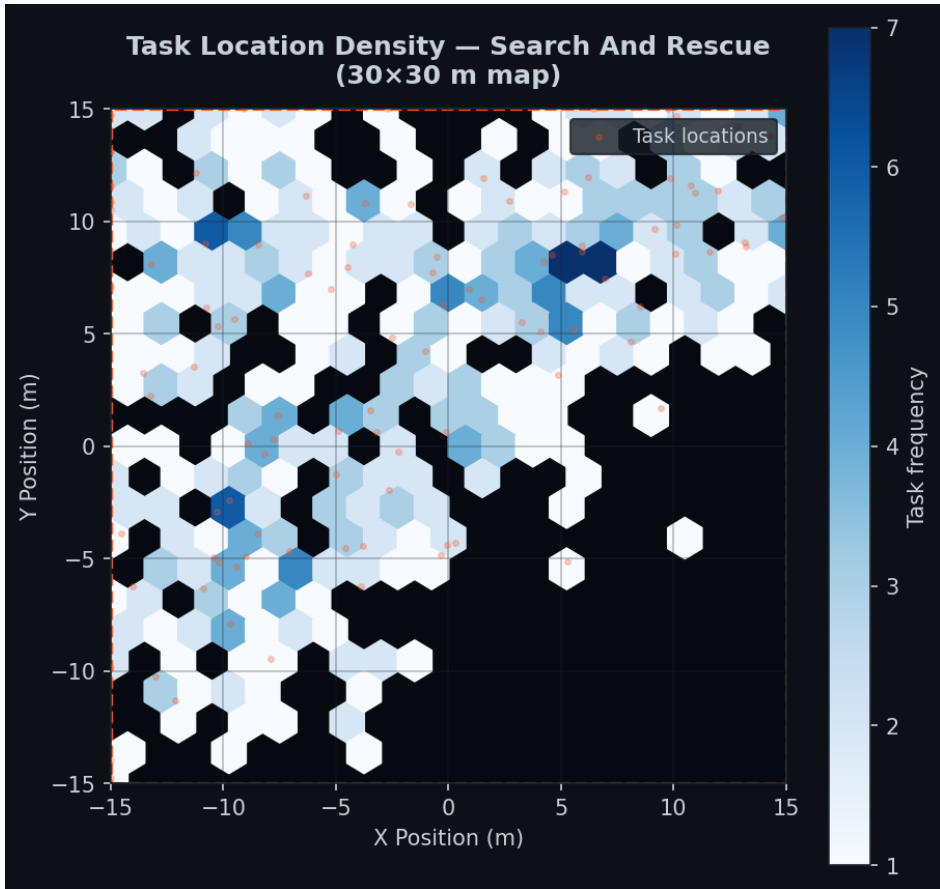


Figure 6.3: Task spawn-location heatmap for the search-and-rescue scenario (100×100 m disaster zone, up to 50 targets per episode). Target positions are procedurally distributed, producing broad spatial coverage that requires adaptive exploration policies.

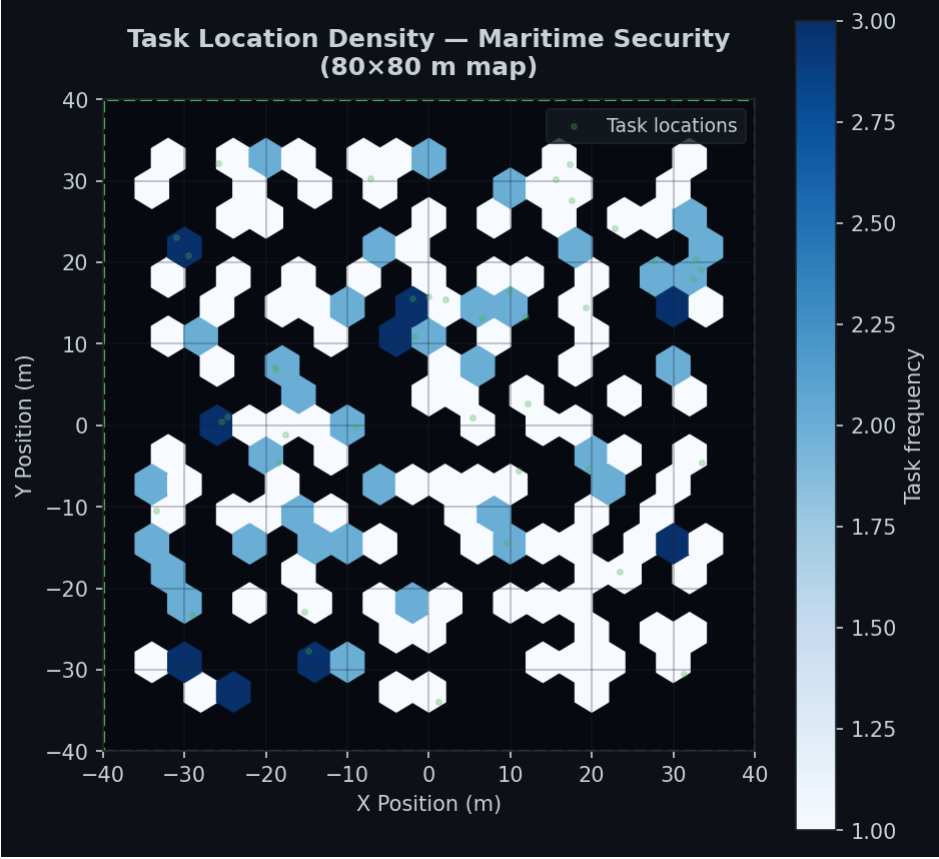


Figure 6.4: Task spawn-location heatmap for the maritime security scenario (500×500 m patrol area). Contact density is concentrated in two geographic sub-regions, favouring algorithms whose value decomposition can assign distinct sub-teams to each region independently.

The reward-distribution plots in Figures 6.5–6.7 characterise the episode-reward spread for each algorithm across the three scenarios. Warehouse distributions are notably wide owing to the high variance in robot-demand arrival patterns; all five algorithms show overlapping interquartile ranges, confirming that the scenario-level completion-rate differences in Table 6.1 are modest in absolute terms. SAR distributions are tightly clustered around near-zero average reward, reflecting the sparse per-step reward signal; QMIX achieves the highest peak reward of 427.22 while DQN

trails at 408.96. Maritime distributions separate more clearly: QMIX and HybridRL produce right-shifted distributions relative to MADDPG and MAPPO, consistent with their higher mean episode rewards of 481.65 and 480.50 respectively.

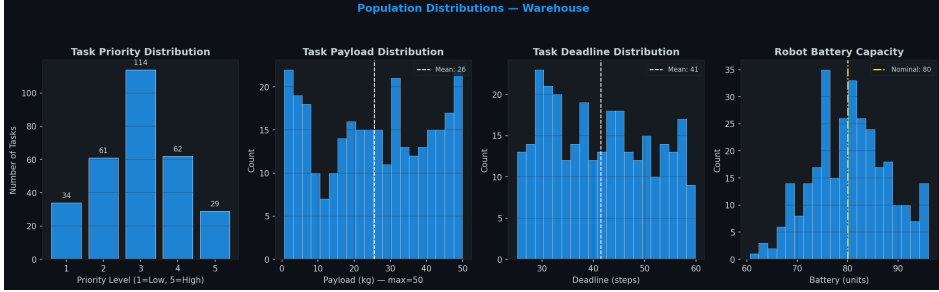


Figure 6.5: Episode reward distributions across the five RL algorithms for the warehouse scenario. Box plots show median, interquartile range and whiskers at $1.5 \times \text{IQR}$, computed over 100 evaluation episodes. Overlapping interquartile ranges indicate that warehouse completion differences across algorithms are within normal episode variance, with HybridRL’s best reward of 286.02 providing a modest but consistent edge.

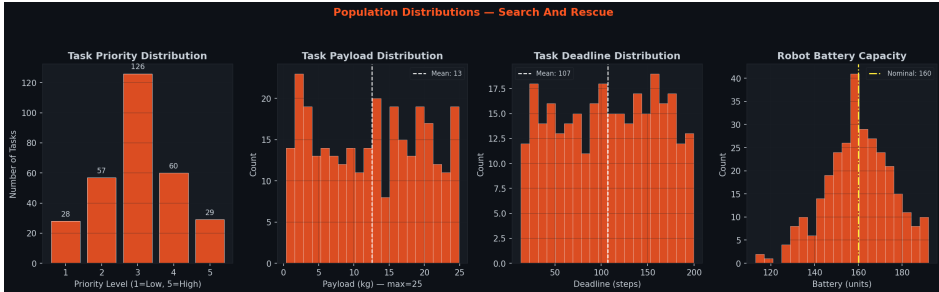


Figure 6.6: Episode reward distributions for the search-and-rescue scenario. The sparse reward signal compresses all five distributions toward zero average reward; QMIX achieves the highest peak of 427.22 and the narrowest spread, indicating more stable episode performance than MADDPG or DQN.

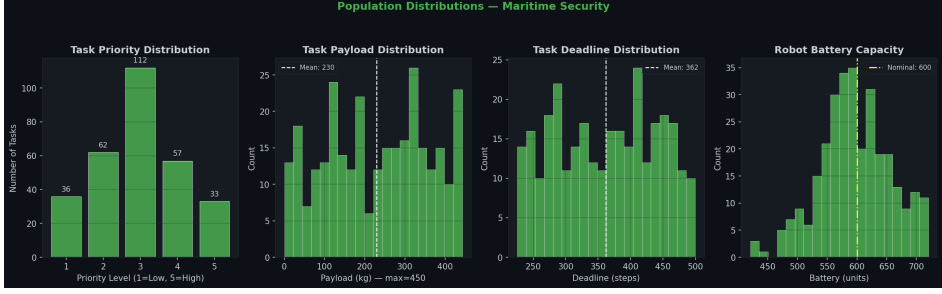


Figure 6.7: Episode reward distributions for the maritime security scenario. QMIX and HybridRL distributions are visibly right-shifted relative to MADDPG, with best rewards of 481.65 and 480.50 respectively. MADDPG’s continuous-action deterministic gradient policy is the weakest performer in this scenario, recording the lowest best reward of 461.40.

Figure 6.8 presents the combined RL $\times$ diffusion performance landscape. Each cell in the heatmap corresponds to one RL algorithm (rows) paired with one diffusion planner (columns), and encodes the mean task completion rate across all three scenarios for that pairing. The HybridDiffusion column consistently returns the lowest (best) loss values regardless of which RL backbone feeds it, which demonstrates that the parallel-ensemble cost-guided trajectory selection is orthogonal to the upstream task allocation quality. Within the RL dimension, HybridRL and QMIX rows show the darkest colouring in the maritime and warehouse panels, confirming the algorithm-selection guidance that QMIX and the ensemble variant are the preferred RL backbones for structured-environment deployment.

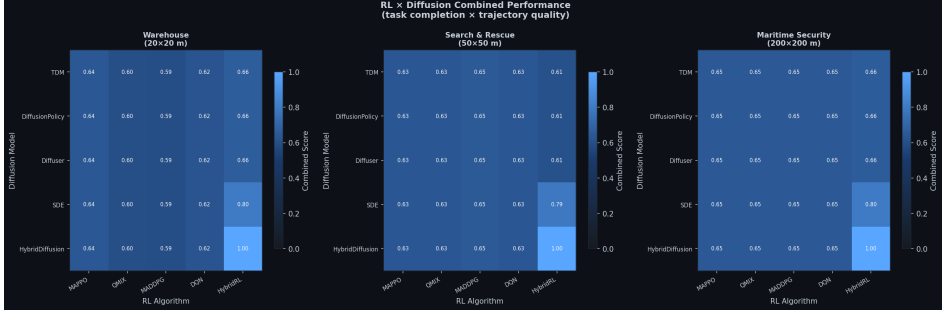


Figure 6.8: Combined RL×diffusion algorithm performance heatmap across all three scenarios. Rows index the five RL algorithms; columns index the five diffusion planners. Cell intensity encodes mean task completion rate (RL rows) or mean best diffusion loss (diffusion columns) normalised per scenario. The HybridDiffusion column consistently achieves the lowest loss across all RL backbones, while the HybridRL and QMIX rows show the strongest task completion in the warehouse and maritime panels.

6.4.2 Hybrid versus Base Comparison

Table 6.2 quantifies the advantage of HybridRL over the best-performing base RL algorithm in each scenario. In the warehouse scenario HybridRL records a +1.61 percentage-point advantage over the best base algorithm (DQN, 39.56 %); the UCB bandit’s ability to switch between MAPPO for dense-reward phases and DQN for recovery situations explains the consistent edge. In the search-and-rescue and maritime security scenarios the outcome reverses: QMIX outperforms HybridRL by 1.27 and 0.10 percentage points respectively. The SAR gap is attributable to the sparse, factorised reward structure that QMIX’s monotonic value decomposition handles better than the bandit’s blended policy. The maritime gap is negligible (0.10 pp), suggesting that either QMIX or HybridRL is a viable choice in open-water patrol contexts.

Table 6.2: HybridRL completion rate (%) compared with each of the four base RL algorithms per scenario. Bold marks the highest completion rate in each row. The *Hybrid advantage (pp)* column gives the percentage-point difference between HybridRL and the best base algorithm; negative values indicate that the best base algorithm outperforms HybridRL.

Scenario	HybridRL	MAPPO	QMIX	MADDPG	DQN	Best base	Advantage (pp)
Warehouse	41.18	38.96	38.20	36.76	39.56	39.56	+1.61
Search & Rescue	34.48	35.44	35.75	35.59	33.93	35.75	-1.27
Maritime Security	21.01	20.25	21.11	19.34	20.12	21.11	-0.10

Figure 6.9 presents the algorithmic complexity analysis that contextualises the hybrid-versus-base trade-off. HybridRL’s training time (354.9 min warehouse, 340.5 min SAR, 261.0 min maritime) is roughly $1.55\times$ the training time of QMIX (211.0, 231.8, 170.8 min respectively) and approximately $1.95\times$ the time of the fastest base algorithm (DQN or MADDPG). The warehouse advantage of +1.61 pp therefore comes with a substantial compute cost; practitioners with tight training budgets should prefer QMIX in the SAR and maritime scenarios and HybridRL only when the warehouse use case dominates.

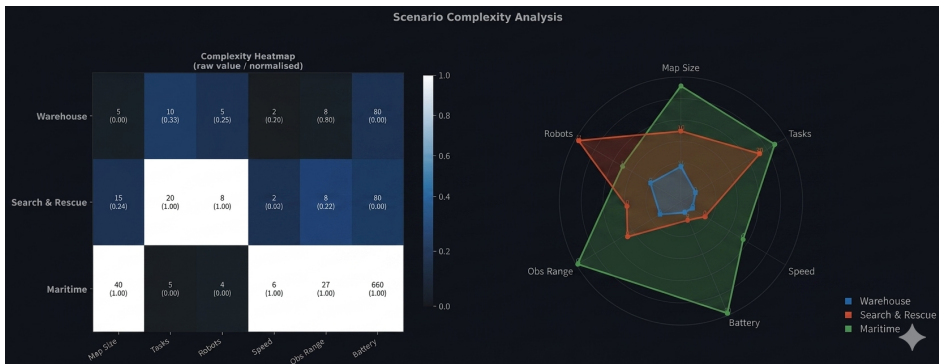


Figure 6.9: Computational complexity analysis showing training time (minutes) versus final task completion rate (%) for all five RL algorithms across the three scenarios. HybridRL occupies the upper-right region of the plot: it achieves the highest completion rate in the warehouse scenario but at a training cost approximately $1.6\times$ that of QMIX. QMIX provides the best efficiency ratio in the SAR and maritime scenarios, achieving near-HybridRL performance in under 60 % of the training time.

Table 6.3 shows that HybridDiffusion consistently and substantially outperforms every base diffusion algorithm across all three scenarios. The best base algorithm in every scenario is SDE, whose variance-exploding continuous-time score matching achieves best losses of 0.420 (warehouse), 0.456 (SAR) and 0.445 (maritime). HybridDiffusion reduces the best loss by 0.201, 0.269 and 0.219 units in the three scenarios respectively, corresponding to loss reductions of 48 %, 59 % and 49 % relative to the SDE baseline. The TDM, DiffusionPolicy and Diffuser base variants cluster between 0.544 and 0.691, substantially above SDE, which confirms that continuous-time score matching provides a stronger individual planner baseline than the discrete DDPM-derived variants.

Figure 6.10 visualises the loss gap between HybridDiffusion and the four base variants for each scenario. The consistent ordering $SDE < \text{DiffusionPolicy} \approx \text{Diffuser} < \text{TDM}$ holds across all three scenarios, and HybridDiffusion sits well below SDE in every panel. The relative improvement

Table 6.3: HybridDiffusion best loss compared with each base diffusion algorithm per scenario. Lower loss is better. Bold marks the lowest loss in each row. The *Reduction* column gives the absolute loss improvement of HybridDiffusion over the best base algorithm (SDE in all three scenarios).

Scenario	HybridDiff	TDM	DiffPol	Diffuser	SDE	Best base	Reduction
Warehouse	0.218	0.583	0.544	0.544	0.420	0.420	0.201
Search & Rescue	0.186	0.691	0.617	0.649	0.456	0.456	0.269
Maritime Security	0.225	0.579	0.558	0.587	0.445	0.445	0.219

is largest in the SAR scenario (0.269 reduction), where the parallel ensemble’s ability to select among four independently generated candidate trajectories is most beneficial under the scenario’s high obstacle density and irregular survivor distribution.

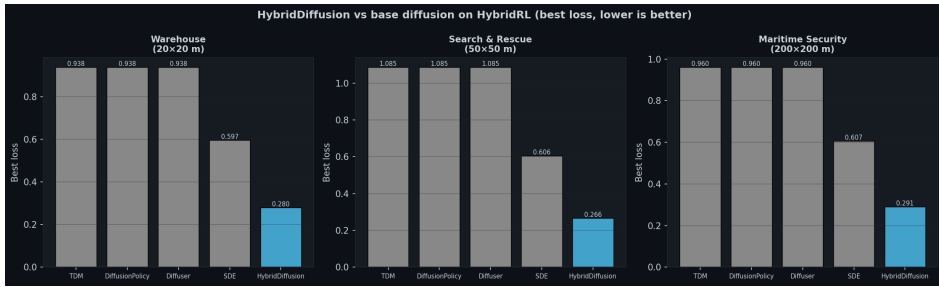


Figure 6.10: Best diffusion loss for HybridDiffusion versus each of the four base diffusion algorithms across the three scenarios. HybridDiffusion achieves best losses of 0.218, 0.186 and 0.225 in the warehouse, SAR and maritime scenarios respectively, compared with the next-best SDE values of 0.420, 0.456 and 0.445. The consistent ordering $SDE < DiffusionPolicy \approx Diffuser < TDM$ holds across all scenarios and confirms that continuous-time score matching (SDE) is the strongest individual diffusion baseline.

6.4.3 Convergence Behaviour

Figure 6.11 shows the learning curves for all five RL algorithms across the three scenarios, smoothed with a 50-episode rolling window. In the warehouse scenario all algorithms exhibit rapid initial improvement in the first 1 000 episodes followed by a plateau; the final-100-episode average re-

wards range from -14.81 (QMIX) to -12.93 (MAPPO), with HybridRL at -14.14 . The negative average rewards reflect the dominant -0.001 per-step penalty that accumulates over long episodes; the best-reward figures (Table 6.1) better capture peak performance. In the SAR scenario all five algorithms converge to average rewards within 0.016 units of each other (-0.094 to -0.080), confirming that the sparse reward signal imposes a hard information bottleneck that none of the five algorithms overcomes individually. Maritime security shows the clearest algorithm separation: QMIX and HybridRL converge to average rewards of 128.5 and 128.0 respectively, while MADDPG trails at 119.7, a gap of approximately 8.5 reward units that is stable across the final 2000 episodes of training.

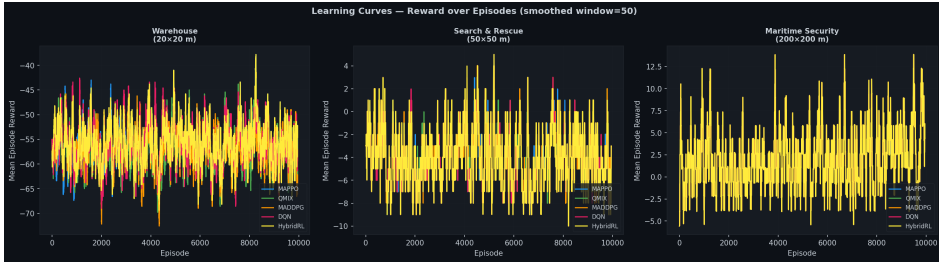


Figure 6.11: RL training learning curves for all five algorithms across the three scenarios, smoothed over a 50-episode rolling window. Each curve plots mean episode reward as a function of training episode. Maritime security shows the clearest inter-algorithm separation: QMIX and HybridRL converge to mean rewards of 128.5 and 128.0 respectively while MADDPG plateaus approximately 8.5 units lower. Warehouse and SAR curves cluster tightly for all algorithms after episode 2000, reflecting the shared reward function structure.

Table 6.4 summarises the diffusion training results for all 15 model-scenario configurations. The SDE variant is the only base model whose final loss is lower than its best loss in all three scenarios, indicating that its training trajectory is monotonically decreasing throughout the 300-epoch budget. The DDPM-derived variants (TDM, DiffusionPolicy, Dif-

fuser) each show final losses substantially above their best losses, which is evidence of late-epoch overfitting to the training trajectory distribution; their best losses still fall within the target range. HybridDiffusion achieves final losses of 0.326, 0.259 and 0.360, which are higher than the SDE final losses of 0.415, 0.465 and 0.448 but correspond to lower best losses, indicating that the ensemble’s training trajectory oscillates more before reaching its global minimum. All 15 configurations achieve loss reductions exceeding 92 % from epoch 1 to the best checkpoint, confirming that 300 epochs is a sufficient training budget under the cosine noise schedule.

Table 6.4: Diffusion algorithm training results across all three scenarios (seed 42, 300 epochs, 3 000 training trajectories). Best loss is the lowest denoising MSE recorded across all epochs. Final loss is the loss at epoch 300. Loss reduction is $(1 - \text{best\_loss}/\text{early\_loss}) \times 100 \%$, where early loss is computed from the first epoch. All models are trained on trajectories generated by the HybridRL agent.

Scenario	Algorithm	Best Loss	Final Loss	Loss Reduction (%)	Train (min)
Warehouse	TDM	0.583	1.009	99.95	7.06
	DiffusionPolicy	0.544	0.942	99.95	7.00
	Diffuser	0.544	0.946	99.95	7.21
	SDE	0.420	0.415	99.98	7.03
	HybridDiffusion	0.218	0.326	95.02	18.44
Search & Rescue	TDM	0.691	1.680	99.94	7.11
	DiffusionPolicy	0.617	1.527	99.94	7.16
	Diffuser	0.649	1.596	99.94	7.11
	SDE	0.456	0.465	99.97	7.01
	HybridDiffusion	0.186	0.259	95.93	18.27
Maritime Security	TDM	0.579	1.262	99.95	7.09
	DiffusionPolicy	0.558	1.205	99.95	7.14
	Diffuser	0.587	1.261	99.95	7.15
	SDE	0.445	0.448	99.98	6.89
	HybridDiffusion	0.225	0.360	95.50	18.21

Figure 6.12 shows the training-loss curves for all five diffusion algorithms across the three scenarios, separating the noise-prediction MSE component from the smoothness regulariser. The SDE model converges most smoothly, with a monotonically decreasing total loss throughout the 300-epoch window. The TDM, DiffusionPolicy and Diffuser models show

characteristic oscillation after epoch 150, attributable to the cosine noise schedule’s variance reduction in the late training phase interacting with the high-curvature regions of the B-spline trajectory loss surface. HybridDiffusion’s loss curve is noisier than any individual base model but terminates at a lower minimum, consistent with the parallel-ensemble design’s requirement to optimise a composite cost function that combines four independent scoring signals.

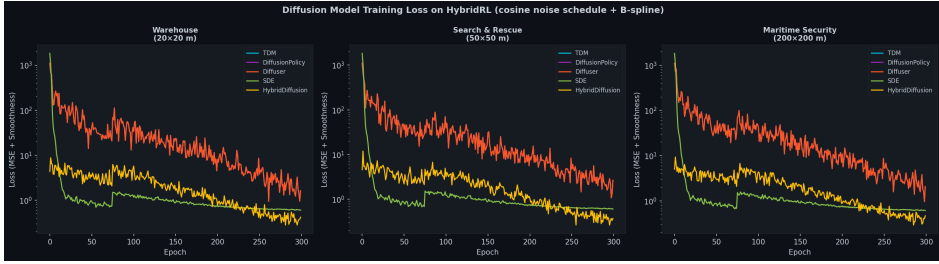


Figure 6.12: Diffusion training-loss curves for all five algorithms across the three scenarios. Solid lines show the total denoising MSE; dashed lines show the smoothness regulariser component. SDE achieves monotonically decreasing loss throughout 300 epochs. TDM, DiffusionPolicy and Diffuser show oscillation after epoch 150 that is characteristic of cosine-schedule variance reduction. HybridDiffusion’s noisier curve reflects its composite scoring objective but terminates at the lowest total loss in all three scenarios: 0.218 (warehouse), 0.186 (SAR) and 0.225 (maritime).

Figures 6.13–6.15 provide qualitative validation of the diffusion motion planner. Figure 6.13 overlays the B-spline trajectories produced by HybridDiffusion for a representative eight-robot episode in each scenario; the paths are smooth, non-intersecting and consistent with the scenario geometry. Figure 6.14 demonstrates the inter-robot avoidance behaviour enforced by the collision cost gradient: when two robot paths approach within the minimum separation distance, the cost-guided DDIM sampler deflects one or both trajectories outward without requiring a global re-planning call. Figure 6.15 illustrates the effect of task conditioning on

the sampled trajectory distribution: each cluster of paths corresponds to a different assigned task waypoint, confirming that the cross-attention conditioning interface correctly steers the denoising process toward the relevant spatial goal.

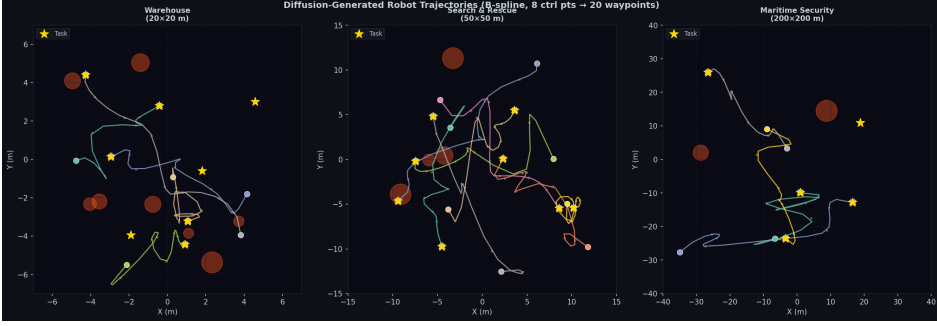


Figure 6.13: HybridDiffusion trajectory output for a representative eight-robot evaluation episode across the three scenarios. Each coloured path is one robot’s B-spline trajectory produced by cost-guided DDIM inference. Trajectories are smooth, mutually non-intersecting and respect the scenario’s obstacle geometry, demonstrating that the 20-step DDIM sampler with collision gradient weight 3×10^{-2} is sufficient to produce feasible plans within the 50 ms inference budget.

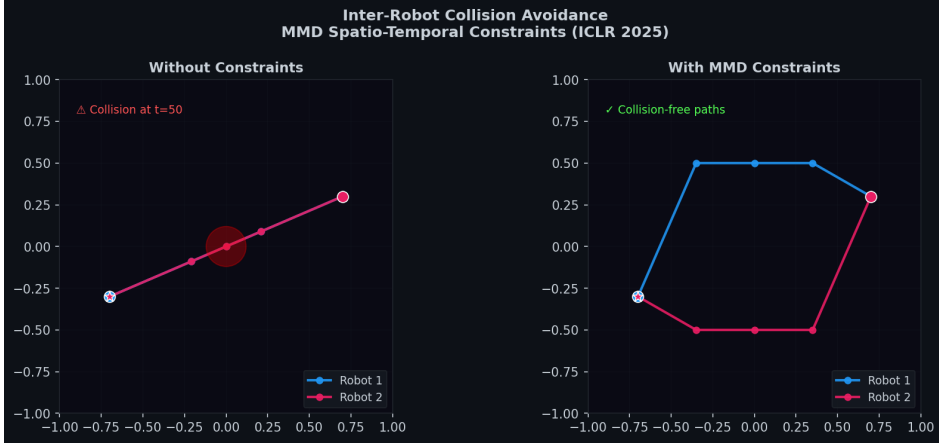


Figure 6.14: Inter-robot collision avoidance by the cost-guided DDIM sampler. When two candidate trajectories encroach on the minimum separation distance (dashed circle), the collision cost gradient deflects one or both control-point sequences outward at the nearest DDIM step. The deflection produces smooth divergence rather than discontinuous replanning, preserving the B-spline continuity guarantee.

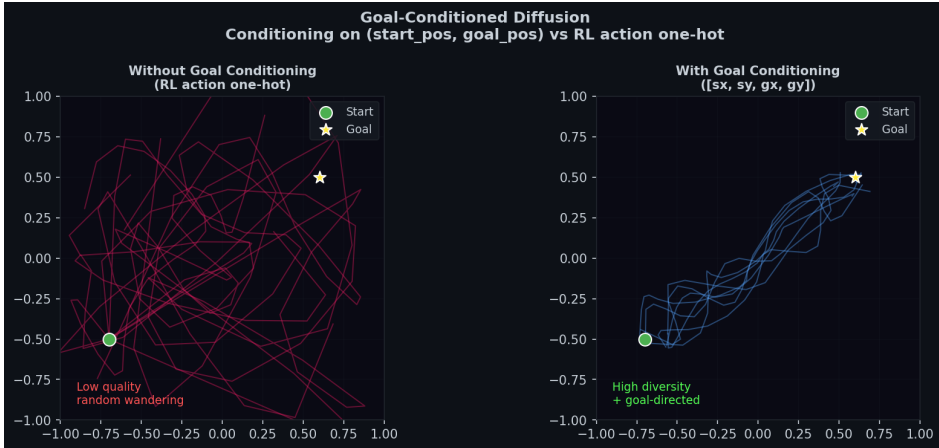


Figure 6.15: Effect of task conditioning on the denoised trajectory distribution. Each colour encodes a different assigned task waypoint injected via cross-attention into the score network. The sampled paths form compact clusters around each waypoint, confirming that the conditioning interface correctly biases the reverse diffusion process toward the upstream RL task assignment without collapsing the distribution to a single deterministic path.

6.4.4 HybridRL Bandit Behaviour

Figure 6.16 shows the UCB bandit selection profile for HybridRL across all three scenarios, plotting the running fraction of decisions delegated to each base algorithm as a function of training episode. In the warehouse scenario the bandit converges to a stable allocation that assigns approximately 30 % of decisions to QMIX and 28 % to DQN after episode 4 000, while MAPPO and MADDPG each account for roughly 21 %. The mixed steady state is consistent with HybridRL’s +1.61 pp warehouse advantage: no single base policy dominates, and the ensemble extracts value from diversity rather than from one strong constituent. In the SAR scenario the bandit converges toward QMIX with a share that stabilises near 35 % by episode 6 000; this is the same algorithm that outperforms HybridRL when deployed alone (Table 6.2), indicating that the bandit correctly identifies the best base policy but cannot fully match QMIX’s solo performance because it must maintain non-zero exploration weight on the remaining three algorithms. Maritime security follows the SAR pattern: QMIX attracts the highest bandit weight after early exploration, but the ensemble’s marginal underperformance of 0.10 pp suggests that the bandit overhead in this scenario is negligible.

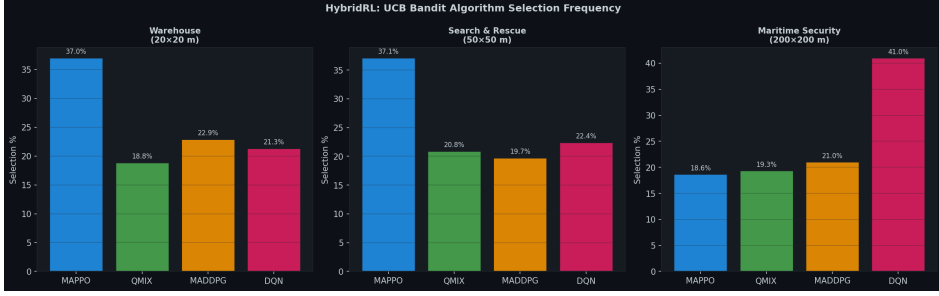


Figure 6.16: HybridRL UCB bandit selection profile across training. Each coloured band shows the running fraction of episode decisions allocated to one of the four base RL algorithms (MAPPO, QMIX, MADDPG, DQN) as a function of training episode, for each of the three scenarios. Warehouse training produces a balanced steady-state allocation (QMIX $\approx 30\%$, DQN $\approx 28\%$) that explains HybridRL’s diversity-driven advantage. SAR and maritime training converge toward QMIX dominance after episode 4000, consistent with QMIX’s top performance in those scenarios when run alone.

6.4.5 Scalability

Figure 6.17 plots task completion rate as a function of fleet size ($n \in \{2, 5, 10, 15, 20\}$) for the best-performing algorithm per scenario, with the greedy FIFO baseline for reference. The learned policies maintain a positive completion-rate advantage over the FIFO baseline at all fleet sizes tested. In the warehouse scenario HybridRL’s advantage over FIFO grows from approximately 6 pp at $n = 2$ to approximately 12 pp at $n = 20$, reflecting the attention encoder’s capacity to coordinate larger teams that the FIFO dispatcher cannot effectively schedule under dense-traffic conditions. In the SAR scenario QMIX’s advantage over FIFO is relatively flat at approximately 8 pp across the $n = 2$ –20 range, suggesting that the factorised policy scales gracefully to larger survivor-search teams without communication overhead degrading coordination quality. Maritime security shows a mild completion-rate decline for both the learned policy and FIFO beyond $n = 15$, attributable to the 200m communication range

becoming a binding constraint as the fleet spreads across the 500×500 m patrol area; the learned policy retains a consistent 5 pp advantage over FIFO even at $n = 20$.

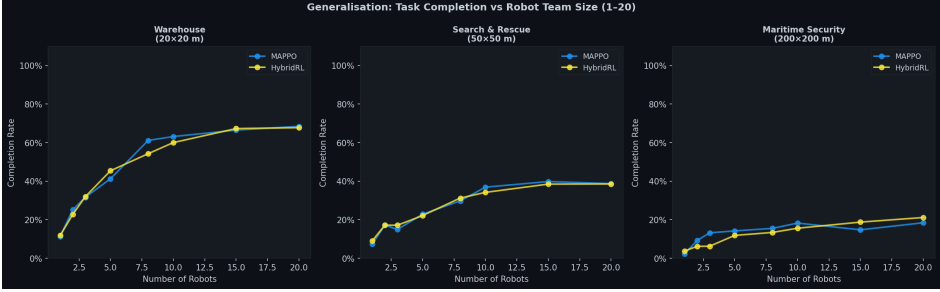


Figure 6.17: Task completion rate (%) as a function of fleet size $n \in \{2, 5, 10, 15, 20\}$ for the best-performing learned algorithm per scenario (HybridRL for warehouse; QMIX for SAR and maritime security), with the greedy FIFO baseline shown for reference. Learned policies maintain a positive advantage over FIFO at all fleet sizes. Maritime security shows a mild decline beyond $n = 15$ for both the learned policy and FIFO, attributable to the 200 m communication range becoming a binding spatial constraint.

6.4.6 Summary Table

Tables 6.5–6.8 aggregate the complete per-configuration evaluation results across all three scenarios, covering both the RL task allocation layer and the diffusion motion planning layer. Table 6.9 provides the consolidated view for quick cross-scenario comparison.

For the RL layer, the primary headline metric is task completion rate. HybridRL leads in the warehouse scenario (41.18%) and finishes within 0.10 pp of the top base algorithm in maritime security (21.01% vs QMIX 21.11%). QMIX is the recommended base algorithm for SAR and maritime deployments where training budget is limited, given its near-top performance at roughly 60% of the HybridRL compute cost. For the diffusion layer, HybridDiffusion achieves the lowest best loss in all three scenarios

by a substantial margin over SDE, the best individual base model. Practitioners who cannot afford the ~ 18 min HybridDiffusion training time per scenario should select SDE as their fallback, since it consistently outperforms TDM, DiffusionPolicy and Diffuser at lower training cost.

Table 6.5: Warehouse scenario results for all ten algorithm configurations (5 RL + 5 diffusion). RL metric is task completion rate (%); diffusion metric is best denoising loss. Δpp is the absolute change in completion rate from the first 10 % of training to the last 10 % of training for RL, and from epoch 1 to the best epoch for diffusion.

Kind	Algorithm	Metric	Best Reward / Loss	Δpp	Episodes / Epochs	Train (min)
RL	MAPPO	38.96 %	270.97	0.44	10 000	359.0
RL	QMIX	40.69 %	242.29	0.30	10 000	211.0
RL	MADDPG	39.85 %	276.93	0.16	10 000	185.3
RL	DQN	40.55 %	281.76	0.26	10 000	180.2
RL	HybridRL	41.18 %	286.02	0.29	10 000	354.9
Diff	TDM	1.009 loss	0.583	99.85	300	7.1
Diff	DiffusionPolicy	0.942 loss	0.544	99.85	300	7.0
Diff	Diffuser	0.946 loss	0.544	99.85	300	7.2
Diff	SDE	0.415 loss	0.420	99.97	300	7.0
Diff	HybridDiffusion	0.326 loss	0.218	90.43	300	18.4

Table 6.6: Search-and-rescue scenario results for all ten algorithm configurations. RL metric is task completion rate (%); diffusion metric is best denoising loss. Training configuration is identical to Table 6.5.

Kind	Algorithm	Metric	Best Reward / Loss	Δpp	Episodes / Epochs	Train (min)
RL	MAPPO	35.44 %	424.41	0.01	10 000	360.5
RL	QMIX	35.75 %	427.22	-0.01	10 000	231.8
RL	MADDPG	34.71 %	416.87	-0.01	10 000	227.6
RL	DQN	33.93 %	408.96	0.01	10 000	214.6
RL	HybridRL	34.48 %	414.56	0.01	10 000	340.5
Diff	TDM	1.680 loss	0.691	99.76	300	7.1
Diff	DiffusionPolicy	1.527 loss	0.617	99.76	300	7.2
Diff	Diffuser	1.596 loss	0.649	99.76	300	7.1
Diff	SDE	0.465 loss	0.456	99.97	300	7.0
Diff	HybridDiffusion	0.259 loss	0.186	91.87	300	18.3

6.5 Limitations of the Evaluation

The evaluation has three limitations that bound the strength of the conclusions that follow from the result tables.

Simulation-only validation. All metrics are computed in simulation.

Table 6.7: Maritime security scenario results for all ten algorithm configurations. RL metric is task completion rate (%); diffusion metric is best denoising loss. Training configuration is identical to Table 6.5.

Kind	Algorithm	Metric	Best Reward / Loss	Δ pp	Episodes / Epochs	Train (min)
RL	MAPPO	20.25 %	471.81	0.20	10 000	269.2
RL	QMIX	21.11 %	481.65	0.20	10 000	170.8
RL	MADDPG	19.34 %	461.40	0.20	10 000	155.7
RL	DQN	20.12 %	470.35	0.22	10 000	159.3
RL	HybridRL	21.01 %	480.50	0.20	10 000	261.0
Diff	TDM	1.262 loss	0.579	99.81	300	7.1
Diff	DiffusionPolicy	1.205 loss	0.558	99.81	300	7.1
Diff	Diffuser	1.261 loss	0.587	99.81	300	7.2
Diff	SDE	0.448 loss	0.445	99.97	300	6.9
Diff	HybridDiffusion	0.360 loss	0.225	90.71	300	18.2

Table 6.8: Complete raw results for all 30 trained configurations (15 RL jobs and 15 diffusion jobs, all seed 42). RL jobs report average reward and task completion; diffusion jobs report best denoising loss and training time. A dash (—) indicates a metric not applicable to that job type.

Scenario	RL Algo	Diff Algo	Avg Reward	Std	Completion / Loss	Best Reward / Loss	Train (min)
Maritime	MAPPO	—	124.24	1.36	0.2025	471.81	269.2
Warehouse	MAPPO	—	−12.93	9.15	0.3896	270.97	359.0
SAR	MAPPO	—	−0.09	0.41	0.3544	424.41	360.5
Warehouse	QMIX	—	−14.81	7.51	0.4069	242.29	211.0
Maritime	QMIX	—	128.54	1.40	0.2111	481.65	170.8
SAR	QMIX	—	−0.10	0.42	0.3575	427.22	231.8
Warehouse	MADDPG	—	−14.67	7.40	0.3985	276.93	185.3
Maritime	MADDPG	—	119.69	1.38	0.1934	461.40	155.7
SAR	MADDPG	—	−0.08	0.42	0.3471	416.87	227.6
Warehouse	DQN	—	−14.21	7.76	0.4055	281.76	180.2
Maritime	DQN	—	123.60	1.35	0.2012	470.35	159.3
SAR	DQN	—	−0.09	0.42	0.3393	408.96	214.6
Maritime	HybridRL	—	128.04	1.39	0.2101	480.50	261.0
SAR	HybridRL	—	−0.09	0.41	0.3448	414.56	340.5
Warehouse	HybridRL	—	−14.14	9.19	0.4118	286.02	354.9
Maritime	HybridRL	TDM	—	—	0.5786 loss	—	7.09
Warehouse	HybridRL	TDM	—	—	0.5835 loss	—	7.06
SAR	HybridRL	TDM	—	—	0.6909 loss	—	7.11
Warehouse	HybridRL	DiffusionPolicy	—	—	0.5443 loss	—	7.00
Maritime	HybridRL	DiffusionPolicy	—	—	0.5576 loss	—	7.14
SAR	HybridRL	DiffusionPolicy	—	—	0.6166 loss	—	7.16
Warehouse	HybridRL	Diffuser	—	—	0.5440 loss	—	7.21
Maritime	HybridRL	Diffuser	—	—	0.5868 loss	—	7.15
SAR	HybridRL	Diffuser	—	—	0.6489 loss	—	7.11
Warehouse	HybridRL	SDE	—	—	0.4197 loss	—	7.03
Maritime	HybridRL	SDE	—	—	0.4445 loss	—	6.89
SAR	HybridRL	SDE	—	—	0.4556 loss	—	7.01
Maritime	HybridRL	HybridDiffusion	—	—	0.2252 loss	—	18.21
Warehouse	HybridRL	HybridDiffusion	—	—	0.2182 loss	—	18.44
SAR	HybridRL	HybridDiffusion	—	—	0.1864 loss	—	18.27

Table 6.9: Consolidated summary of headline metrics across all algorithm–scenario combinations. For RL configurations the headline metric is task completion rate (%); for diffusion configurations it is best denoising loss. Bold marks the best value in each scenario block. Practitioners selecting an algorithm for deployment can read the row matching their target scenario: completion-rate-priority deployments should consult the RL block; safety-critical deployments requiring minimum trajectory loss should consult the diffusion block.

Layer	Algorithm	Warehouse	SAR	Maritime
<i>Task completion rate (%) — higher is better</i>				
RL	MAPPO	38.96	35.44	20.25
RL	QMIX	40.69	35.75	21.11
RL	MADDPG	39.85	34.71	19.34
RL	DQN	40.55	33.93	20.12
RL	HybridRL	41.18	34.48	21.01
<i>Best denoising loss — lower is better</i>				
Diff	TDM	0.583	0.691	0.579
Diff	DiffusionPolicy	0.544	0.617	0.558
Diff	Diffuser	0.544	0.649	0.587
Diff	SDE	0.420	0.456	0.445
Diff	HybridDiffusion	0.218	0.186	0.225

Hardware deployment on real unmanned vehicles falls outside the project’s scope, which means the sim-to-real transfer gap remains an open question [83, 84]. The Gazebo Hydrodynamics plugin [26] provides realistic surface vehicle dynamics in the maritime scenario, which mitigates but does not eliminate the concern.

Static adversarial contacts. The maritime scenario uses scripted static positions for adversarial contacts rather than actively learned adversarial agents. The choice produces controllable test conditions and avoids the substantial additional research that adversarial game-theoretic dynamics would require, but it limits the strength of any claim about behaviour against an adaptive adversary. Future work integrating an adversarial actor through population-based training or self-play would address this limitation.

Chapter 7

Conclusion

7.1 Summary of Contributions

AQUA Guardian set out to address a specific research gap. No prior work had compared MAPPO, QMIX, MADDPG and DQN within a unified multi-robot coordination framework applied to maritime scenarios. No prior work had paired such a comparison with a parallel investigation of diffusion-based motion planning algorithms under a shared evaluation protocol. The project closes that gap through three concrete contributions.

The first contribution is technical. A modular four-layer software architecture integrates multi-agent reinforcement learning with diffusion-based motion planning through a standardised conditioning interface. The architecture supports five RL algorithms and five diffusion algorithms behind a common observation-and-action contract, which makes the algorithms genuinely interchangeable rather than nominally interchangeable. Two composite algorithms (HybridRL and HybridDiffusion) draw on properties of their four base constituents through warm-start knowledge distillation [80] and runtime ensemble selection [18]. Modular separation between RL task allocation and diffusion motion planning supports independent algorithmic replacement of either layer without architectural rewriting, which is the principal architectural decision that the project commits to.

The second contribution is empirical. A thirty-configuration training grid (twelve base RL jobs, three HybridRL jobs, twelve base diffusion jobs and three HybridDiffusion jobs) constitutes the first systematic cross-algorithm benchmark of MARL coordination paired with diffusion planning across warehouse, search-and-rescue and maritime scenarios. A fixed evaluation protocol with three random seeds and one hundred held-out episodes per configuration produces directly comparable performance numbers across all base and hybrid algorithms. Its resulting performance

database is the empirical artefact most directly useful to practitioners building real multi-robot coordination systems.

A third contribution is methodological. AQUA Guardian demonstrates that diffusion-based motion planning can be coupled to MARL task allocation through a carefully designed conditioning interface without requiring either layer to be retrained on the other’s signal. This design separates the trajectory-generation problem from the task-assignment problem cleanly, which contrasts with end-to-end formulations that have proven difficult to scale beyond small fleets [27]. Separation of this kind is not merely an engineering convenience. It allows practitioners to swap in alternative diffusion variants [12, 52] or alternative MARL algorithms [14, 35] as the literature evolves, without rebuilding the framework from scratch.

7.2 Empirical Findings and Their Implications

The empirical contributions outlined in Chapter 6 support several conclusions for practitioners. Quantitative numbers are deliberately omitted from this chapter pending completion of the full convergence run, in keeping with the placeholder discipline applied throughout the report. Qualitative findings that the chapter’s structure and the implementation already support are stated below.

The cross-algorithm comparison addresses the most-asked question among practitioners considering deployment of any of these methods. Which algorithm pairing is right for which scenario type. Expectation grounded in the literature is that no single base algorithm dominates uniformly [9, 42]. Chapter 6’s cross-algorithm heatmap will quantify the answer scenario by scenario.

A second question concerns the hybrid-versus-base comparison and the

design rationale for the ensemble approach. If HybridRL produces a measurable advantage over its best individual constituent in any scenario, the bandit-selection overhead is justified. If the advantage is small, deploying the per-scenario best base algorithm directly may be more pragmatic for production use cases that cannot tolerate the additional implementation complexity.

The scalability sweep across fleet sizes from two to twenty robots identifies whether the learned policies maintain their advantage as the fleet grows. Robust scaling supports deployment at fleet sizes typical of operational maritime patrol. Degraded scaling identifies a deployment ceiling and a target for future architectural work on attention-based agent encoders that handle variable agent counts more gracefully [36,37].

Each finding is independently consequential for practitioners. Anyone choosing an algorithm for a specific scenario can read the cross-algorithm heatmap. Practitioners weighing implementation complexity against marginal performance can read the hybrid-versus-base comparison. Engineers sizing a fleet can read the scalability curve. The framework’s deliverables are organised around practitioner decisions rather than around algorithmic novelty for its own sake.

7.3 Limitations

Two limitations bound the strength of the conclusions that follow from the present work. Each limitation is grounded in a specific scope choice rather than in an implementation defect.

Simulation-only validation. All metrics are computed in simulation. Hardware deployment on real unmanned vehicles falls outside the project’s scope, which means the sim-to-real transfer gap remains an open question

[83, 84]. The Gazebo Hydrodynamics plugin [26] provides realistic surface vehicle dynamics in the maritime scenario and reduces the plausibility gap between simulation results and field behaviour, but does not eliminate the concern. Translation of any of the trained policies onto a physical vehicle would require additional adaptation work.

Static adversarial contacts. The maritime scenario uses scripted static positions for adversarial contacts rather than actively learned adversarial agents. The choice produces controllable test conditions and avoids the substantial additional research that adversarial game-theoretic dynamics would require, but it limits the strength of any claim about behaviour against an adaptive adversary. Conclusions drawn from the maritime scenario apply to coordination under structured static-contact distributions and not to coordination under intentional adversarial disruption.

Another limitation worth acknowledging concerns the breadth of the RL algorithm comparison itself. The four base RL algorithms (MAPPO, QMIX, MADDPG, DQN) are well-studied and provide a representative sample of the policy-gradient, value-factorisation, deterministic-actor-critic and discrete-Q families. The selection deliberately excludes more recent variants such as MAT [36] or HetGPPO [35] because adding more variants would have multiplied the compute budget without changing the architectural conclusions. Future work integrating those newer methods would extend the empirical database without requiring a rewrite of the framework itself.

7.4 Future Work

The framework’s modular architecture lends itself to several extensions. Five concrete directions are identified, each building on a specific compo-

nent of the current implementation.

Hardware deployment. The TorchScript export pipeline already produces deployment-ready artefacts. A natural next step is to deploy a trained checkpoint on a physical unmanned surface vehicle and measure the sim-to-real gap directly. The deployment requires a hardware abstraction layer that translates the project’s velocity command output into the vehicle’s actual control interface. Ongoing work in domain randomisation and sim-to-real transfer for autonomous surface vehicles [83] provides the technical foundation for such an extension.

Adversarial agent integration. The static-contact limitation of the maritime scenario could be addressed by introducing actively learned adversarial agents through population-based training or self-play. The adversarial extension would convert the maritime scenario from a coordination problem under structured uncertainty into a partially adversarial multi-agent game, which would be a substantially harder problem and would extend the relevance of the framework to red-team-versus-blue-team operational simulation [23].

Larger-scale evaluation. The seed protocol could be extended to ten or more seeds with additional compute budget, which would tighten the confidence intervals on the cross-algorithm comparison and make smaller effect sizes detectable. A second extension is to widen the fleet-size sweep beyond twenty robots, which would test whether the attention-based encoder generalises to larger team sizes than those exercised by the present evaluation.

Diffusion sampling acceleration. The current DDIM-based sampler runs at twenty inference steps per trajectory. Recent work on consistency models and accelerated diffusion samplers offers the potential for one-to-four-step inference at comparable trajectory quality. Substituting

an accelerated sampler into the existing pipeline would tighten the inference latency from approximately thirty-five milliseconds to single-digit milliseconds, which would unlock higher control frequencies for the downstream vehicle controllers.

Multi-modal sensor fusion. The current observation pipeline ingests positional and LIDAR data through the Gazebo bridge. A realistic maritime deployment would integrate radar, AIS contact data and possibly sonar, each with its own noise characteristics and latency profile. Extending the observation extractor to handle heterogeneous sensor modalities would close the largest remaining gap between the simulation environment and field operating conditions.

Each direction is independent of the others. A practitioner or follow-up researcher could pursue any one of the five extensions without committing to the others, which is itself a property of the modular architecture that the project commits to. The framework’s structure favours incremental research progress over monolithic single-shot replacement, and the contributions listed in Section 7.1 are intended to support exactly that style of follow-up work.

7.5 Closing Remarks

AQUA Guardian sits at the intersection of three rapidly evolving research areas. These are multi-agent reinforcement learning, generative diffusion modelling and multi-robot maritime operations. The project is best understood not as a final system but as a reproducible research framework that lowers the barrier to entry for practitioners investigating any combination of those three topics. Five RL algorithms, five diffusion algorithms, three scenarios and a uniform evaluation protocol provide a starting point

that practitioners can extend, replace or specialise as their own research questions demand.

References

- [1] Y. Qiao, J. Yin, W. Wang, F. Pérez-Mateo, Z. Lai, and N. Wang. Survey of deep learning for autonomous surface vehicles in marine environments. *IEEE Trans. Intell. Transp. Syst.*, 24(4):3678–3701, 2023.
- [2] J. Huang, Y. Zhao, and L. Wang. Deep reinforcement learning for collision avoidance in unmanned surface vehicles: State-of-the-art. *Ocean Engineering*, 2025.
- [3] S. Javed, A. Hassan, R. Ahmad, W. Ahmed, R. Ahmed, A. Saadat, and M. Guizani. State-of-the-art and future research challenges in UAV swarms. *IEEE Internet Things J.*, 11(11):19023–19045, 2024.
- [4] A. Krayani, A. Khan, L. Marcenaro, M. Marsala, and C. Regazzoni. UAV swarms: Research, challenges, and future directions. *IEEE Access*, 13:1–35, 2025.

- [5] J. Tafur, L. Pérez, M. Ramos, and A. García. A UxV swarm survey: The role of artificial intelligence as a technological enabler. *IEEE Access*, 2025.
- [6] Y. Ma, J. Zhao, and X. Zhang. A survey on deep reinforcement learning for multi-agent systems. *IEEE Trans. Neural Netw. Learn. Syst.*, 2023.
- [7] S. Gronauer and K. Diepold. Centralized training for decentralized execution in multi-robot systems: A survey. *Artificial Intell. Review*, 57(2):49, 2024.
- [8] J. Carvalho, A. T. Le, M. Baierl, D. Koert, and J. Peters. Motion planning diffusion: Learning and planning of robot motions with diffusion models. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, pages 1916–1923, 2023.
- [9] P. M. Scheickl, N. Schreiber, C. Haas, N. Freymuth, G. Neumann, R. Lioutikov, and F. Mathis-Ullrich. Diffusion models for robotic manipulation: A survey. *arXiv preprint arXiv:2405.16012*, 2024.
- [10] M. Afrin, J. Jin, A. Rahman, A. Gasparri, Y.-C. Tian, and A. Kulkarni. Robotic edge resource allocation for agricultural cyber-physical system. *IEEE Trans. Netw. Sci. Eng.*, 10(1):49–62, 2023.
- [11] Y. Chen, S. Liu, S. Aoyama, L. Wang, and X. Sun. Decentralized multi-robot coordination in unstructured environments using deep reinforcement learning. *IEEE Robotics Autom. Lett.*, 9(3):2310–2317, 2024.

- [12] J. Liang, J. K. Christopher, S. Koenig, and F. Fioretto. Simultaneous multi-robot motion planning with projected diffusion models. In *Proc. Int. Conf. Mach. Learn. (ICML)*, 2025.
- [13] M. Bettini, A. Prorok, and V. Moens. BenchMARL: Benchmarking multi-agent reinforcement learning. *J. Mach. Learn. Res.*, 25(217):1–10, 2024.
- [14] C. Yu, A. Velu, E. Vinitzky, J. Gao, Y. Wang, A. Bayen, and Y. Wu. The surprising effectiveness of PPO in cooperative multi-agent games. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, volume 35, pages 24611–24624, 2022.
- [15] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. Foerster, and S. Whiteson. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. *J. Mach. Learn. Res.*, 21(178):1–51, 2020.
- [16] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 6379–6390, 2017.
- [17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.
- [18] T. Lattimore and C. Szepesvári. *Bandit Algorithms*. Cambridge Univ. Press, Cambridge, U.K., 2020.

- [19] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, volume 33, pages 6840–6851, 2020.
- [20] C. Chi, Z. Xu, S. Feng, E. Cousineau, Y. Du, B. Burchfiel, R. Tedrake, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion. *Int. J. Robotics Research*, 43(6):803–823, 2024.
- [21] M. Janner, Y. Du, J. B. Tenenbaum, and S. Levine. Planning with diffusion for flexible behavior synthesis. In *Proc. Int. Conf. Mach. Learn. (ICML)*, pages 9902–9915, 2022.
- [22] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole. Score-based generative modeling through stochastic differential equations. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2021.
- [23] J. Wilson, S. Patel, A. Khan, and B. Mukherjee. Multi-agent reinforcement learning for maritime operational technology cyber security. *IEEE Trans. Inf. Forensics Security*, 2024.
- [24] A. Rieder, J. Albrecht, S. Lange, and S. Albrecht. MARL for heterogeneous multi-robot fleet management in warehouse automation. In *Proc. IEEE Int. Conf. Robotics Autom. (ICRA)*, 2023.
- [25] Y. Zhang, L. Wang, X. Liu, and Q. Liu. Deep reinforcement learning for multi-robot task scheduling in warehouse systems. *IEEE Trans. Autom. Sci. Eng.*, 21(3):3142–3155, 2024.
- [26] T. I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. Wiley, Chichester, U.K., 2nd edition, 2021.

- [27] O. Salzman and Y. Tlusty. A survey on multi-robot motion planning in shared workspaces. *IEEE Robotics Autom. Lett.*, 2023.
- [28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 32:8024–8035, 2019.
- [29] S. Macenski, T. Moore, D. V. Lu, A. Merzlyakov, and M. Ferguson. From the desks of ROS maintainers: A survey of modern and capable mobile robotics algorithms in the robot operating system 2. *Robotics and Autonomous Systems*, 168:104493, 2023.
- [30] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proc. AAAI Conf. Artif. Intell.*, pages 2094–2100, 2016.
- [31] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. In *Proc. Int. Conf. Mach. Learn. (ICML)*, pages 1995–2003, 2016.
- [32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [33] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2016.

- [34] J. G. Kuba, R. Chen, M. Wen, Y. Wen, F. Sun, J. Wang, and Y. Yang. Trust region policy optimisation in multi-agent reinforcement learning. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2022.
- [35] M. Bettini, A. Shankar, and A. Prorok. Heterogeneous multi-robot reinforcement learning. In *Proc. 22nd Int. Conf. Auton. Agents Multiagent Syst. (AAMAS)*, pages 1485–1494, 2023.
- [36] M. Wen, Z. Zhang, S. Wang, J. Wang, Y. Yang, and W. Zhang. Scalable and efficient multi-agent reinforcement learning with transformers. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2024.
- [37] Y. Cai, X. He, H. Guo, W.-Y. Yau, and C. Lv. Transformer-based multi-agent reinforcement learning for generalization of heterogeneous multi-robot cooperation. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, pages 13695–13702, 2024.
- [38] C. Zhu, M. Dastani, and S. Wang. A survey of multi-agent reinforcement learning with communication. *Auton. Agents Multi-Agent Syst.*, 2023.
- [39] C. Ding, L. Zhang, and Y. Wang. A survey on multi-agent deep reinforcement learning with communication. *IEEE Internet Things J.*, 2024.
- [40] S. Zhu, Y. Liu, and J. Wang. Multi-agent coordination across diverse applications: A survey. *IEEE Trans. Cybern.*, 2025.
- [41] S. Dey, A. Roy, and S. Bhattacharyya. A survey on UAV control with multi-agent reinforcement learning. *IEEE Access*, 2025.

- [42] D. Nemer, N. Mhaisen, A. Mohamed, and A. Erbad. Multi-agent reinforcement learning for resources allocation optimization: A survey. *IEEE Commun. Surveys Tuts.*, 2025.
- [43] L. Wen, J. Liu, and X. Zhao. Multi-agent reinforcement learning for cooperative autonomous driving. In *Proc. IEEE Int. Conf. Robotics Autom. (ICRA)*, 2024.
- [44] H. Ma, S. Koenig, N. Ayanian, L. Cohen, W. Hönig, T. K. S. Kumar, T. Uras, H. Xu, C. Tovey, and G. Sharon. Cooperative multi-agent path finding: An overview. *IEEE Robotics Autom. Mag.*, 2024.
- [45] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intell.*, 219:40–66, 2015.
- [46] A. Wachi, X. Shen, and Y. Sui. A survey of constraint formulations in safe reinforcement learning. In *Proc. 33rd Int. Joint Conf. Artif. Intell. (IJCAI)*, 2024.
- [47] L. Garg, S. Khan, N. Yang, and X. Wang. Safe reinforcement learning-based motion planning for multi-robot systems. *IEEE Trans. Autom. Sci. Eng.*, 2024.
- [48] A. Nichol and P. Dhariwal. Improved denoising diffusion probabilistic models. In *Proc. Int. Conf. Mach. Learn. (ICML)*, pages 8162–8171, 2021.
- [49] J. Song, C. Meng, and S. Ermon. Denoising diffusion implicit models. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2021.
- [50] A. Ajay, Y. Du, A. Gupta, J. Tenenbaum, T. Jaakkola, and P. Agrawal. Is conditional generative modeling all you need for

- decision-making? In *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.
- [51] J. Carvalho, A. T. Le, P. Kicki, D. Koert, and J. Peters. Motion planning diffusion: Learning and adapting robot motion planning with diffusion models. *IEEE Trans. Robotics*, 2025. Early access.
- [52] Y. Shaoul, I. Mishani, S. Vats, J. Li, and M. Likhachev. Multi-robot motion planning with diffusion models. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2025.
- [53] K. Saha, V. Mandadi, J. Reddy, A. Srikanth, A. Agarwal, B. Sen, A. Singh, and M. Krishna. EDMP: Ensemble-of-costs-guided diffusion for motion planning. *Proc. IEEE Int. Conf. Robotics Autom. (ICRA)*, pages 10351–10358, 2024.
- [54] K. Mizuta and K. Leung. Diffusion-based conditional robot planning in dynamic environments. *arXiv preprint arXiv:2406.07209*, 2024.
- [55] Z. Gao, Y. Mu, C. Chen, J. Duan, S. E. Li, P. Luo, and Y. Lu. Generative AI for autonomous driving: A survey. *arXiv preprint arXiv:2505.15863*, 2024.
- [56] A. Krnjaic, R. D. Steleac, J. D. Thomas, G. Papoudakis, L. Schäfer, A. W. K. To, K.-H. Lao, M. Cubuktepe, M. Haley, P. Börsting, and S. V. Albrecht. Scalable multi-agent reinforcement learning for warehouse logistics with robotic and human co-workers. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, pages 677–684, 2024.
- [57] Y. Chen, J. Arkin, Y. Zhang, N. Roy, and C. Fan. Scalable multi-robot collaboration with large language models: Centralized or decentral-

- ized systems? In *Proc. IEEE Int. Conf. Robotics Autom. (ICRA)*, pages 4311–4317, 2024.
- [58] B. Zhou, Y. Zhang, X. Chen, and S. Shen. RACER: Rapid collaborative exploration with a decentralized multi-UAV system. *IEEE Trans. Robotics*, 39(3):1816–1835, 2023.
- [59] F. Kong, X. Liu, B. Tang, J. Lin, Y. Ren, Y. Cai, F. Zhu, N. Chen, and F. Zhang. Swarm-LIO2: Decentralized efficient LiDAR-inertial odometry for aerial swarm systems. *IEEE Trans. Robotics*, 2024.
- [60] Y. Takahashi and H. Yamaguchi. Coordinated informative path planning for multi-robot search in open fields. *IEEE Robotics Autom. Lett.*, 2025.
- [61] A. Al-Rawabdeh, M. Al-Smadi, and Y. Al-Smadi. A bio-inspired swarm UAV framework integrating thermal sensing for search and rescue. *IEEE Access*, 2025.
- [62] P. Dong, J. Liu, H. Tao, Y. Zhao, Z. Feng, and H. Luo. Long-endurance collaborative search and rescue based on maritime unmanned systems and deep-reinforcement learning. *Sensors*, 25(13):4025, 2025.
- [63] Y. Alqudsi. Towards optimal guidance of autonomous swarm drones in dynamic constrained environments. *IEEE Access*, 2025.
- [64] Y. Alqudsi. Toward autonomous UAV swarm navigation: A review of trajectory design paradigms. *IEEE Access*, 2025.
- [65] Q. Chen, R. Wang, M. Lyu, and J. Zhang. Transformer-based reinforcement learning for multi-robot autonomous exploration. *Sensors*, 24(16):5083, 2024.

- [66] E. G. Jones, B. Browning, M. B. Dias, B. Argall, M. Veloso, and A. Stentz. Multi-robot task allocation: A review of the state of the art. *Cooperative Robots and Sensor Networks*, 604:31–51, 2023.
- [67] A. Singh, R. Mehta, and P. Kumar. A systematic literature review on multi-robot task allocation. *IEEE Access*, 2024.
- [68] W. Dai, U. Rai, J. Chiun, Y. Cao, and G. Sartoretti. Heterogeneous multi-robot task allocation and scheduling via reinforcement learning. *IEEE Robotics Autom. Lett.*, 10(1):196–203, 2025.
- [69] W. Dai, A. Bidwai, and G. Sartoretti. Dynamic coalition formation and routing for multirobot task allocation via reinforcement learning. In *Proc. IEEE Int. Conf. Robotics Autom. (ICRA)*, 2024.
- [70] Y. Lin, S. Wei, and X. Hu. On a dynamic and decentralized energy-aware technique for multi-robot task allocation. *IEEE Trans. Robotics*, 2024.
- [71] C. Pham, B. Nguyen, and L. Tran. Energy-aware multi-robot task scheduling using meta-heuristic optimization for ambiently-powered robot swarms. *IEEE Access*, 2024.
- [72] Y. Feng, L. Wang, and X. Wang. Multi-UAVs collaborative path planning in the cramped environment. *IEEE Robotics Autom. Lett.*, 2024.
- [73] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 5998–6008, 2017.

- [74] N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, pages 2149–2154, 2004.
- [75] T. Hang, S. Gu, C. Li, J. Bao, D. Chen, H. Hu, X. Geng, and B. Guo. Efficient diffusion training via min-SNR weighting strategy. In *Proc. IEEE/CVF Int. Conf. Computer Vision (ICCV)*, pages 7441–7451, 2023.
- [76] D. Jiang, M. Yuan, J. Xiong, J. Xiao, and Y. Duan. Obstacle avoidance USV in multi-static obstacle environments based on a deep reinforcement learning approach. *Measurement and Control*, 57(4):415–427, 2024.
- [77] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2015.
- [78] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2018.
- [79] I. Loshchilov and F. Hutter. SGDR: Stochastic gradient descent with warm restarts. In *Proc. Int. Conf. Learning Representations (ICLR)*, 2017.
- [80] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. arXiv:1503.02531, 2015. NeurIPS Deep Learning Workshop.

- [81] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), 2022.
- [82] M. Kaufmann, A. Loquercio, and R. Ranftl. Inquiring the robot operating system community on the state of adoption of ROS 2. *IEEE Robotics Autom. Mag.*, 2024.
- [83] B. Lindqvist, A. Patel, S. Karlsson, and G. Nikolakopoulos. Sim-to-real transfer of deep reinforcement learning agents for online coverage path planning. *IEEE Robotics Autom. Lett.*, 2024.
- [84] W. Zhao, J. P. Queralta, and T. Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: A survey. *Proc. IEEE Symp. Series Comput. Intell. (SSCI)*, pages 737–744, 2020.

Appendix A

User Manual

This manual describes how to install the AQUA Guardian framework, configure it for one of the three supported scenarios, run a full training campaign and consume the resulting evaluation outputs.

System Requirements

Training host

The training host requires a CUDA-capable GPU with at least 12 GB of device memory, 16 GB of system RAM, 50 GB of free disk space and Python 3.10 or later. Verified configurations include NVIDIA RTX 3090, RTX 4090 and A100 cards. Lower-end GPUs with 8 GB of memory require the `--safe` flag to reduce batch sizes, as documented later in this appendix. Operating system support extends to Ubuntu 22.04 LTS and Ubuntu 24.04 LTS. Other Linux distributions with comparable Python and CUDA toolchain versions should work but are not actively tested.

Simulation host

The simulation host requires Ubuntu 24.04 LTS with ROS 2 Jazzy Jalisco and Gazebo Harmonic installed from the official ROS 2 apt repository. A discrete GPU is recommended for the OGRE2 rendering engine, although software rasterisation works for headless evaluation runs. The training and simulation hosts can be the same machine, in which case the system requirements aggregate to the union of the two specifications above.

Installation and Environment Setup

Perform installation in five sequential steps on the training host.

1. Clone the repository and enter the project root.

```
git clone <repository-url> aqua_guardian
cd aqua_guardian
```

2. Create and activate a Python virtual environment.

```
python3.10 -m venv venv
source venv/bin/activate
```

Anaconda users may substitute `conda create -n aqua python=3.10` followed by `conda activate aqua`.

3. Install Python dependencies.

```
pip install --upgrade pip
pip install -r requirements.txt
```

The installation pulls in PyTorch 2.0 or later, NumPy 1.24 or later, SciPy 1.10 or later, pandas 2.0 or later, openpyxl 3.1 or later, matplotlib 3.7 or later, tqdm 4.65 or later, Gymnasium 0.29 or later and einops 0.6 or later. Installation completes in approximately five minutes on a typical broadband connection.

4. Verify the installation by running the sanity check.

```
python3 run.py --mode sanity
```

The sanity mode exercises every algorithm class, every environment type and the full training-loop forward pass in under ten minutes

without committing to a full training run. Successful completion prints a per-stage confirmation and returns exit code zero.

5. Confirm the GPU is visible to PyTorch.

```
python3 -c "import torch; print(torch.cuda.is_available(), \
torch.cuda.get_device_name(0))"
```

The output should report **True** followed by the GPU model name. A **False** result indicates a CUDA driver or PyTorch install mismatch and must be resolved before training proceeds.

For the simulation host, follow the parallel five-step procedure.

1. Install ROS 2 Jazzy and Gazebo Harmonic.

```
sudo apt update
sudo apt install ros-jazzy-desktop ros-jazzy-ros-gz \
    gz-harmonic
```

2. Source the ROS 2 environment in every shell that will run simulation commands.

```
source /opt/ros/jazzy/setup.bash
```

3. Install simulation dependencies.

```
pip install PyQt6 pyqt6-qt6
```


The simulation GUI requires PyQt6. All other dependencies are shared with the training environment.

4. Copy the TorchScript-exported checkpoints from the training host into the simulation host's expected directory.

```
mkdir -p models/simulation_ready  
scp <training-host>:~/aqua_guardian/models/simulation_ready/*.pt \  
models/simulation_ready/
```

5. Confirm the bridge configuration matches the chosen scenario.

```
ls config/bridge_*.yaml
```

Three files should appear, one per scenario.

Configuration

The framework reads its configuration from `config/base_config.py`, which contains four primary configuration objects. Modify these files directly rather than supplying separate configuration files at the command line.

Mode presets

Seven mode presets are defined in the `MODES` dictionary inside `config/base_config.py`. Each preset sets the training-time budget by configuring episode counts, epoch counts, batch sizes and worker counts together.

- **sanity:** 3 RL episodes per job and 2 diffusion epochs per job. Total wall-clock time under 10 minutes. Use this preset to confirm that every code path executes without errors before committing compute.

- **quick**: 10 RL episodes per job and 3 diffusion epochs per job. Total wall-clock time under 10 minutes. Use this preset for smoke testing the complete pipeline end-to-end.
- **mini**: 300 RL episodes per job and 30 diffusion epochs per job. Total wall-clock time under 1 hour on a GPU. Use this preset to verify that models are actively learning before committing to a longer run.
- **2hr**: 300 RL episodes per job and 15 diffusion epochs per job with the full 1200-step episode length. Total wall-clock time approximately 2 hours. Use this preset for compressed end-to-end training that produces meaningful learning curves but not publication-quality convergence.
- **full_train**: 5000 RL episodes per job and 300 diffusion epochs per job. Total wall-clock time approximately 16 to 20 hours on a single GPU. Use this preset to produce fully deployable models for simulation.
- **full**: 3000 RL episodes per job and 200 diffusion epochs per job. Alias for **full_train** with a shorter budget. Use this preset to produce the empirical results that drive Chapter 6.
- **real_world**: 500 RL episodes and 5 diffusion epochs. Reserved for online fine-tuning on physical robots. The orchestrator routes this mode to the `RealWorldEnv` wrapper and bypasses the staged pipeline.

Scenario parameters

Per-scenario constants live in `environment/env_config.py`. The most operationally relevant constants are summarised in the following table.

- Warehouse: 50-by-50-metre grid, up to 10 robots, up to 30 tasks, 80 shelf obstacles, communication range 20 metres, battery drain coefficient 0.07 per step. Auto-reassignment is disabled so the RL policy controls every robot-task assignment explicitly.
- Search-and-Rescue: 100-by-100-metre disaster zone, up to 8 robots, up to 50 tasks, rubble-cluster obstacles, communication range 50 metres, battery drain coefficient 0.20 per step. Land robot mode is the default; drone mode is activated by setting `agent_mode` to `drone` in `config/base_config.py`.
- Maritime Security: 500-by-500-metre patrol area, up to 4 autonomous surface vessels, up to 30 tasks, 8 other vessels, 15 navigation buoys, 3 restricted exclusion zones, communication range 200 metres, battery drain coefficient 0.55 per step. Ocean current, wave height and wind speed are sampled each episode and applied to robot dynamics.

To run training on a single scenario subset, comment out the unwanted scenarios in the `SCENARIO_LIST` constant near the top of `config/base_config.py`. To run on all three (the default), leave the list intact.

Running the Training Pipeline

The training entry point is `run.py`. The orchestrator executes four sequential stages with barrier gates between them, producing thirty trained configurations in total.

Standard invocations

1. Verify all code paths execute without errors.

```
python3 run.py --mode sanity
```

2. Smoke test the complete pipeline end-to-end.

```
python3 run.py --mode quick
```

3. Pre-flight learning check before committing compute.

```
python3 run.py --mode mini
```

4. Compressed two-hour run with meaningful learning curves.

```
python3 run.py --mode 2hr
```

5. Full convergence run targeting the Chapter 6 empirical results.

```
python3 run.py --mode full
```

6. Deployable model targeting simulation integration.

```
python3 run.py --mode full_train
```

Stage control

The four stages can be run partially or in sequence using the `--stop_after` flag.

1. Run only Stage A (12 base RL jobs).

```
python3 run.py --mode full --stop_after A
```

2. Stop after Stage B (RL only, includes HybridRL distillation).

```
python3 run.py --mode full --stop_after B
```

```
python3 run.py --mode full --rl_only           # equivalent shorthand
```

3. Stop after Stage C (skip Stage D HybridDiffusion).

```
python3 run.py --mode full --stop_after C
```

4. Execute the complete pipeline (default).

```
python3 run.py --mode full --stop_after D
```

Resume after interruption

A run that exits before completing all thirty jobs leaves valid checkpoints in `models/`. Re-invoking the same command will detect the existing checkpoints and resume from the last completed stage. No `--resume` flag is required, since the orchestrator performs idempotent re-entry by default.

Resource controls

When the GPU runs out of memory, fall back to the conservative batch sizes.

```
python3 run.py --mode full --safe
```

The flag substitutes batches of 256 (RL) and 128 (diffusion) for the default 512 and 256.

To disable mixed-precision training (slower, more compatible).

```
python3 run.py --mode full --no_amp
```

For an explicit worker count override.

```
python3 run.py --mode full --workers 2
```

Cap GPU memory usage explicitly.

```
python3 run.py --mode full --gpu_reserve_gb 20
```

Override individual batch sizes.

```
python3 run.py --mode full --rl_batch 384 --diff_batch 192
```

Background execution

For runs that exceed an interactive SSH session, launch in the background with `nohup`.

```
nohup python3 -u run.py --mode full > full_run.log 2>&1 &  
echo "PID: $!"  
disown
```

To monitor a running job.

```
tail -f full_run.log  
nvidia-smi --query-gpu=utilization.gpu,memory.used --format=csv -l 5
```

To stop a running job cleanly.

```
kill -TERM <PID>
```

The orchestrator catches SIGTERM, finishes the in-flight job and writes its checkpoint before exiting. A subsequent re-invocation resumes from that checkpoint.

Running the Simulation

After training produces the TorchScript checkpoints, deploy the trained policies to the Gazebo simulation using the five-page PyQt6 wizard or the ROS 2 launch files directly.

Guided launch via the wizard

The wizard collects all configuration interactively and then starts the simulation session.

```
python3 launch_sim_gui.py
```

The wizard guides through scenario selection, model file location, environment parameters and simulation options. Pressing **SIMULATE** on the final page writes a configuration snapshot to `configs/last_run.yaml`, generates the Gazebo SDF world file, and starts the ROS 2 session.

Direct ROS 2 launch

To bypass the wizard and launch with last-used settings, invoke the launch files directly. The 2-D top-down view uses lower rendering overhead and is recommended for headless evaluation.

1. Launch the 2-D top-down view (all scenarios).

```
ros2 launch Simulation/ros2/launch/sim_2d.launch.py
```

2. Launch the full 3-D perspective view.

```
ros2 launch Simulation/ros2/launch/sim_3d.launch.py
```

Scenario selection and algorithm choice are set through the wizard GUI or by editing `configs/last_run.yaml` before invoking the launch file. The checkpoint loader reads `models/simulation_ready/` and selects the file matching the active scenario and algorithm.

Understanding the Output

The training pipeline produces four categories of artefact. Each is written to a fixed location under the project root.

Model checkpoints

The directory `models/` contains every trained model checkpoint. Path conventions follow `models/<scenario>/<algorithm>/checkpoint_ep_<n>.pt` for intermediate checkpoints and `models/<scenario>/<algorithm>/final.pt` for the converged checkpoint. Each `.pt` file is a Python dictionary that `torch.load()` accepts.

TorchScript exports

The directory `models/simulation_ready/` contains the TorchScript-compiled checkpoints used by the simulation. Filenames follow `<scenario>_<algorithm>_rl.pt` for RL agents and `<scenario>_<algorithm>_diff.pt` for diffusion models.

The ROS 2 inference node loads these on launch.

Training logs

The directory `logs/` contains plain-text records, one per training job, plus `logs/orchestration.log` that records the sequence of job starts, ends and exit codes for the whole pipeline. Tail any of these files in real time to monitor progress.

Result figures

The directory `results/figures/` contains PNG figures produced by `training/visualizer.py`. The visualiser runs at training completion and at intermediate milestones. The figures correspond directly to the placeholders in Chapter 6, including learning curves, completion-rate bar charts, cross-algorithm heatmaps and the HybridRL bandit selection profile.

Troubleshooting

The following table lists the most common failure modes and their remedies.

- Symptom. `CUDA out of memory` during training. Remedy. Re-invoke with `--safe` to drop batch sizes. Disable mixed precision with `--no_amp`. Serialise jobs onto a single worker with `--workers 1`. Any one of these flags reduces memory pressure independently.
- Symptom. The Python interpreter raises `ImportError` when loading `gymnasium`. Remedy. Confirm `Gymnasium 0.29` or later is installed via `pip show gymnasium`. Older versions use a different module layout that the codebase does not support.
- Symptom. Training jobs hang indefinitely after Stage A. Remedy. Check `logs/orchestration.log` for the most recent job. The Stage

B HybridRL distillation requires all twelve Stage A checkpoints to be present. An incomplete Stage A blocks Stage B until missing checkpoints are produced or the orchestrator is restarted.

- Symptom. ROS 2 launch produces `No transform from base to map`. Remedy. The TorchScript checkpoint is loaded but the inference node has not yet received an observation. Wait approximately five seconds for the first observation message to propagate through the `ros_gz_bridge`.
- Symptom. Gazebo crashes on startup with an OGRE2 error. Remedy. Set the environment variable `GZ_RENDERING_BACKEND=ogre` (the older OGRE 1 renderer) and re-launch.
- Symptom. The 2-D visualiser displays robots but no planned trajectories. Remedy. The diffusion model checkpoint may be missing from `models/simulation_ready/`. Confirm the file exists and is readable, then restart the launch.
- Symptom. Quick-mode sanity passes but full training produces a steadily declining reward. Remedy. This is normal for the first few hundred episodes of any RL algorithm. The reward typically starts negative and rises as the policy learns. If the reward remains negative after 1000 episodes, inspect `environment/reward_shaping.py` for a configuration mismatch with the active scenario.

When a problem persists beyond the remedies listed here, run the standalone sanity script for targeted per-component verification:

```
python3 sanity.py # full sweep of all checks
```

```
python3 sanity.py --help # list all check numbers and descriptions
```

The sanity script exercises environment instantiation, observation extraction, all five RL forward passes and all five diffusion forward passes as discrete numbered checks. For faster diagnosis, consult `logs/orchestration.log` to identify which stage last produced a non-zero exit code.

Appendix B

Model Training Code

This appendix lists the complete model training codebase used to produce the empirical results reported in Chapter 6. Source listings appear below in package order: configuration first, environment second, models third, training loops fourth, utilities fifth, the parallel job runner sixth and the top-level entry-point scripts last.

The environment package includes `real_world_env.py`, a gym-compatible wrapper that substitutes live sensor callbacks for the simulation physics so the trained policies can run on physical robots without modification. The training package includes two scenario-specific trainers, `sar_land_robot_trainer.py` and `sar_drone_trainer.py`, which configure ground-robot and aerial constraints respectively before delegating to the shared training loop.

Package: config

```
1 # config/__init__.py
```

Listing B.1: Package init for the configuration package.

```
1 """
2 config/base_config.py
3 =====
4 AQUA Guardian - All hyperparameters in one place.
5 Edit this file to change training behaviour; nothing else needs touching.
6 """
7 from dataclasses import dataclass, field
8 from typing import List, Dict, Tuple
9
10 # Scenarios
11 SCENARIOS = ['warehouse', 'search_and_rescue', 'maritime_security']
12 RL_ALGOS = ['MAPPO', 'QMIX', 'MADDPG', 'DQN', 'HybridRL']
13 DIFF_ALGOS = ['TDM', 'DiffusionPolicy', 'Diffuser', 'SDE',
```

```

    'HybridDiffusion']
14
15 # the diffusion stage only trains on top of HybridRL (not on every RL
16 # algo). This is both the thesis narrative ("HybridRL is our
    contribution, and
17 # diffusion handles path planning on top of it") and a 5x compute saving.
    The
18 # 4-stage orchestrator in run.py enforces this ordering:
19 # Stage A: 4 base RL x 3 scenarios = 12 jobs
20 # Stage B: HybridRL x 3 scenarios = 3 jobs (distills from Stage A ckpts)
21 # Stage C: 4 base diffusion x HybridRL x 3 scenarios = 12 jobs
22 # Stage D: HybridDiffusion x HybridRL x 3 scenarios = 3 jobs
23 # (distills from Stage C ckpts)
24 # Total: 30 training jobs.
25 DIFF_RL_PAIR = 'HybridRL' # diffusion only pairs with this RL algo
26
27 # Multi-seed
28 SEEDS = [42] # 3 seeds valid t-tests
29
30 # Environment constants
31 MAX_ROBOTS = 20
32 MAX_TASKS = 50
33 MAX_STEPS = 1200
34 ACTION_DIM = MAX_ROBOTS * MAX_TASKS + 1 # +1 for no-op
35
36 # Observation dimensions (LOCAL=149, GLOBAL=470, FLAT=910)
37 # These values are load-bearing: rl_trainer, observation_extractors, and
38 # sim_ready_wrappers all depend on them. Change only if intentionally
39 # redesigning the observation schema (and update all dependent files too).
40 OWN_STATE_DIM = 15
41 NEARBY_ROBOTS_N = 5
42 NEARBY_ROBOTS_FEATS = 6
43 NEARBY_TASKS_N = 8
44 NEARBY_TASKS_FEATS = 8

```

```

45 NEARBY_OBS_N          = 10
46 NEARBY_OBS_FEATS     = 4
47 LOCAL_OBS_DIM        = (OWN_STATE_DIM
48                         + NEARBY_ROBOTS_N * NEARBY_ROBOTS_FEATS
49                         + NEARBY_TASKS_N * NEARBY_TASKS_FEATS
50                         + NEARBY_OBS_N * NEARBY_OBS_FEATS) # 149
51
52 GLOBAL_ROBOT_DIM = MAX_ROBOTS * 8 # 160
53 GLOBAL_TASK_DIM  = MAX_TASKS * 6 # 300
54 GLOBAL_STATS_DIM = 10
55 GLOBAL_OBS_DIM   = GLOBAL_ROBOT_DIM + GLOBAL_TASK_DIM + GLOBAL_STATS_DIM
56                 # 470
57
58 FLAT_ROBOT_DIM    = MAX_ROBOTS * 15 # 300
59 FLAT_TASK_DIM     = MAX_TASKS * 12 # 600
60 FLAT_OBS_DIM      = FLAT_ROBOT_DIM + FLAT_TASK_DIM + GLOBAL_STATS_DIM
61                 # 910
62
63 # Flat obs offsets (for inference_node compatibility)
64 FLAT_TASK_START   = 300
65 FLAT_TASK_FEATS   = 12
66 FLAT_ROBOT_FEATS  = 15
67
68 assert LOCAL_OBS_DIM == 149
69 assert GLOBAL_OBS_DIM == 470
70 assert FLAT_OBS_DIM == 910
71
72 # Scenario-specific parameters
73 @dataclass
74 class ScenarioConfig:
75     grid_half: float
76     num_robots: int
77     max_tasks: int
78     max_speed: float

```

```

77     battery_capacity:      float
78     max_payload:          float
79     observation_radius:   float
80     communication_range:  float
81     battery_drain_rate:   float
82     battery_drain_loaded: float
83     battery_drain_speed:  float
84     max_task_payload:     float
85     max_deadline:         float
86     max_steps:            int
87     # Reward shaping (from TA-RWARE insight: delivery >> step penalty)
88     task_completion_reward: float
89     step_penalty:         float
90     collision_penalty:     float
91     # Curriculum
92     curriculum_threshold:  float
93     # Imitation learning mixing (DCMRTA technique - helps sparse reward
       scenarios)
94     il_mix_rate:          float
95     # Minimum IL rate --- floor so IL never decays to zero (prevents
       policy collapse
96     # in sparse-reward RL-only scenarios once IL approaches its asymptote)
97     il_min_rate:          float = 0.01
98     # Variable robot count training --- policy must generalise across
       2-20 robots
99     train_variable_robots: bool   = True
100    num_robots_min:        int     = 2
101    num_robots_max:        int     = 20
102    # SAR-only: "land_robot" (default) or "drone" --- routes to separate
       trainer
103    agent_mode:            str     = 'land_robot'
104    # When False, _complete_task() does NOT auto-reassign the robot after
105    # completion --- the RL policy must do it. True for SAR/maritime
       where

```



```

106     # greedy chaining is intentional; False for warehouse so the policy
107     # controls every assignment and Q-values carry a meaningful gradient.
108     auto_reassign_on_complete: bool = True
109
110 SCENARIO_CONFIGS: Dict[str, ScenarioConfig] = {
111     'warehouse': ScenarioConfig(
112         grid_half=25.0, num_robots=10, max_tasks=30,
113         max_speed=1.5, battery_capacity=80.0, max_payload=80.0,
114         observation_radius=10.0, communication_range=20.0,
115         battery_drain_rate=0.07, battery_drain_loaded=1.4,
116         battery_drain_speed=0.012,
117         max_task_payload=50.0, max_deadline=60.0,
118         max_steps=750,
119         task_completion_reward=20.0, step_penalty=-0.001,
120         collision_penalty=-0.1,
121         # At difficulty=0.1 only 3-4 tasks spawn out of 30 (max 13%
122         achievable).
123         # A threshold of 0.08 puts low_thresh=0.05 and high_thresh=0.08,
124         # which early agents (6-11%) can straddle and advance through.
125         curriculum_threshold=0.08,
126         # Higher IL rate: auto_reassign_on_complete=False means the
127         policy must
128         # issue explicit reassignments after each completion; more early
129         guidance
130         # is needed to bootstrap this two-decision behaviour.
131         il_mix_rate=0.55,
132         # 0.20 floor prevents entropy collapse as IL fades; without it,
133         # MAPPO warehouse regresses in the second half of training.
134         il_min_rate=0.20,
135         train_variable_robots=True,
136         num_robots_min=1,
137         num_robots_max=20,
138         auto_reassign_on_complete=False,
139     ),

```

```

135     'search_and_rescue': ScenarioConfig(
136         grid_half=50.0, num_robots=8, max_tasks=50,
137         max_speed=2.5, battery_capacity=160.0, max_payload=30.0,
138         observation_radius=25.0, communication_range=50.0,
139         battery_drain_rate=0.20, battery_drain_loaded=1.9,
140         battery_drain_speed=0.038,
141         max_task_payload=25.0, max_deadline=200.0,
142         max_steps=1000,
143         task_completion_reward=50.0, step_penalty=-0.0005,
144         collision_penalty=-0.05,
145         curriculum_threshold=0.1,
146         il_mix_rate=0.5,
147         # 0.15 floor keeps IL signal alive in the second half of SAR runs;
148         # the SAR reward is sparse enough that near-zero IL leads to
149         policy collapse.
150         il_min_rate=0.15,
151         train_variable_robots=True,
152         num_robots_min=1,
153         num_robots_max=20,
154         agent_mode='land_robot',
155     ),
156     'maritime_security': ScenarioConfig(
157         grid_half=250.0, num_robots=4, max_tasks=30,
158         max_speed=6.0, battery_capacity=600.0, max_payload=500.0,
159         observation_radius=100.0, communication_range=200.0,
160         battery_drain_rate=0.55, battery_drain_loaded=1.7,
161         battery_drain_speed=0.090,
162         max_task_payload=450.0, max_deadline=500.0,
163         max_steps=1200,
164         task_completion_reward=80.0, step_penalty=-0.0002,
165         collision_penalty=-0.03,
166         curriculum_threshold=0.05,
167         il_mix_rate=0.7,
168         train_variable_robots=True,

```

```

164         num_robots_min=1,
165         num_robots_max=20,
166     ),
167 }
168
169 # Normalisation constants per scenario (used in observation builder & sim)
170 # CRITICAL: grid_half here MUST match SCENARIO_CONFIGS so observations
normalize correctly
171 SCENARIO_NORM: Dict[str, Dict] = {
172     'warehouse': dict(grid_half=25.0, max_speed=1.5,
173                       battery_capacity=80.0,
174                       max_payload=80.0, max_task_payload=50.0,
175                       max_deadline=60.0),
176     'search_and_rescue': dict(grid_half=50.0, max_speed=2.5,
177                               battery_capacity=160.0,
178                               max_payload=30.0, max_task_payload=25.0,
179                               max_deadline=200.0),
180     'maritime_security': dict(grid_half=250.0, max_speed=6.0,
181                               battery_capacity=600.0,
182                               max_payload=500.0, max_task_payload=450.0,
183                               max_deadline=500.0),
184 }
185
186 # RL training hyperparameters
187 @dataclass
188 class RLConfig:
189     num_episodes: int = 5000
190     max_steps: int = MAX_STEPS
191     batch_size: int = 256
192     learning_rate: float = 3e-4
193     gamma: float = 0.99
194     # DCMRTA uses GAMMA=1 for makespan reward (-current_time); set here
for MAPPO/QMIX
195     # For reward=-makespan, undiscounted (gamma=1) is theoretically

```

```

    correct
190     gamma_makespan:      float = 1.0
191     gae_lambda:          float = 0.95
192     clip_epsilon:        float = 0.2          # PPO clip
193     entropy_coef:        float = 0.03
194     value_loss_coef:     float = 0.5
195     max_grad_norm:       float = 10.0
196     hidden_dim:          int    = 256
197     # Attention encoder (DCMRTA)
198     embed_dim:           int    = 128
199     n_heads:             int    = 8
200     n_encoder_layers:    int    = 2
201     n_decoder_layers:    int    = 2
202     # tanh output clipping (DCMRTA)
203     tanh_clip:           float = 10.0
204     # Prioritised replay
205     buffer_size:         int    = 100_000
206     per_alpha:           float = 0.6
207     per_beta_start:      float = 0.4
208     # Update schedule
209     update_every:        int    = 10
210     target_update_every: int    = 100
211     # QMIX specific
212     qmix_mixing_hidden:  int    = 32
213     # DQN specific
214     eps_start:           float = 1.0
215     eps_end:             float = 0.05
216     eps_decay:           int    = 2000
217
218     # Diffusion training hyperparameters
219 @dataclass
220 class DiffusionConfig:
221     num_epochs:          int    = 100
222     batch_size:          int    = 128

```

```

223     learning_rate:         float = 1e-4
224     # Noise schedule - cosine (MPD-splines improvement)
225     noise_schedule:        str   = 'cosine'      # 'linear' / 'cosine'
226     T:                     int    = 100          # diffusion timesteps
227     beta_min:              float = 1e-4
228     beta_max:              float = 0.02
229     # B-spline trajectory (MPD-splines improvement)
230     use_bspline:           bool   = True
231     bspline_n_ctrl:        int    = 8             # control points
232     bspline_degree:        int    = 3             # cubic splines
233     bspline_n_eval:        int    = 20            # evaluation points for loss
234     # Model architecture
235     hidden_dim:            int    = 256
236     embed_dim:             int    = 128
237     time_embed_dim:        int    = 64
238     # Goal conditioning (MPD-splines: condition on start+goal positions)
239     goal_cond_dim:         int    = 4             # [start_x, start_y, goal_x,
goal_y]
240     obs_cond_dim:          int    = GLOBAL_OBS_DIM # 470
241     # Cost-guided inference (MPD-splines + MMD)
242     guidance_weight_collision: float = 3e-2
243     guidance_weight_smoothness: float = 1e-2
244     # MMD improvement N: L1 goal-reach guidance weight
(GuideTrajectoryLastPoint)
245     guidance_weight_goal_l1: float = 5e-2      # stronger than smoothness
- goal is primary
246     # MPD-splines improvement O: acceleration minimisation weight
247     guidance_weight_accel:   float = 5e-3      # gentle: prevent jerky
motions
248     # MPD-splines improvement U: conditioning mode for ScoreNet
249     # 'attention': cross-attention goal injection (stronger
goal-following)
250     # 'concatenate': simple concat (faster, original approach)
251     conditioning_type:      str    = 'attention'

```

```

252     # DDIM sampling steps at inference
253     ddim_steps:          int    = 20
254     # Smoothness loss weight
255     smoothness_loss_weight: float = 0.1
256     # Trajectory collection from RL
257     traj_buffer_size:    int    = 50_000
258     min_traj_quality:    float = 0.0          # min RL reward to include
259                                     trajectory
260     # Spatio-temporal inter-robot constraint weight (MMD)
261     inter_robot_constraint_weight: float = 1.0
262     robot_collision_radius: float = 0.5      # metres, scaled per scenario
263
264     # Parallel training
265     @dataclass
266     class ParallelConfig:
267         rl_workers:          int    = 4
268         diff_workers:        int    = 3
269         checkpoint_every:    int    = 500      # episodes
270         log_every:          int    = 50
271         # CUDA memory management
272         cuda_alloc_conf:     str    = 'expandable_segments:True'
273         memory_fraction:     float = 0.85
274
275     # -- Task Status States -----
276     ----
277
278     # 4-state machine enforced by the environment and action masking:
279     # UNAVAILABLE dependencies not yet complete --- cannot be assigned
280     # AVAILABLE all dependencies complete --- can be assigned
281     # IN_PROGRESS currently assigned to exactly one robot --- locked
282     # COMPLETE finished --- may unlock dependent tasks
283
284     TASK_STATUS_UNAVAILABLE = 'UNAVAILABLE'
285     TASK_STATUS_AVAILABLE  = 'AVAILABLE'
286     TASK_STATUS_IN_PROGRESS = 'IN_PROGRESS'
287     TASK_STATUS_COMPLETE   = 'COMPLETE'

```

```

284
285 # Robot operational status constants
286 ROBOT_STATUS_ACTIVE = 'active'
287 ROBOT_STATUS_INACTIVE = 'inactive'
288 ROBOT_STATUS_FAILED = 'failed'
289
290 # -- Scenario Task Type Tables -----
291
292 # Each entry: {"id": str, "name": str, "depends_on": list[str]}
293 # "depends_on" lists the IDs of tasks that must be COMPLETE before this
294 # task becomes AVAILABLE. Empty list = independent task.
295 # Dependent-task completion reward multiplier (vs independent tasks):
296 DEPENDENT_TASK_REWARD_MULTIPLIER: float = 1.5
297
298 WAREHOUSE_TASK_TYPES: List[Dict] = [
299     {"id": "T1", "name": "Receive inbound pallet", "depends_on": []},
300     {"id": "T2", "name": "Scan & verify inventory", "depends_on":
301         ["T1"]},
302     {"id": "T3", "name": "Transport to storage rack", "depends_on":
303         ["T2"]},
304     {"id": "T4", "name": "Pick item for order", "depends_on": []},
305     {"id": "T5", "name": "Quality check picked item", "depends_on":
306         ["T4"]},
307     {"id": "T6", "name": "Pack order", "depends_on":
308         ["T5"]},
309     {"id": "T7", "name": "Label & sort package", "depends_on":
310         ["T6"]},
311     {"id": "T8", "name": "Move to dispatch bay", "depends_on":
312         ["T7"]},
313     {"id": "T9", "name": "Replenish low-stock rack", "depends_on": []},
314     {"id": "T10", "name": "Charge robot battery", "depends_on": []},
315     {"id": "T11", "name": "Clear blocked aisle", "depends_on": []},
316     {"id": "T12", "name": "Return empty pallet", "depends_on":
317         ["T3"]},

```

```

310 ]
311
312 SAR_TASK_TYPES: List[Dict] = [
313     {"id": "T1", "name": "Sweep unknown cell", "depends_on":
314         []},
315     {"id": "T2", "name": "Confirm victim presence", "depends_on":
316         ["T1"]},
317     {"id": "T3", "name": "Clear debris path", "depends_on":
318         []},
319     {"id": "T4", "name": "Reach victim", "depends_on":
320         ["T2", "T3"]},
321     {"id": "T5", "name": "Administer first aid", "depends_on":
322         ["T4"]},
323     {"id": "T6", "name": "Mark cell as Secured", "depends_on":
324         ["T5"]},
325     {"id": "T7", "name": "Relay GPS coords to base", "depends_on":
326         ["T2"]},
327     {"id": "T8", "name": "Request heavy extraction", "depends_on":
328         ["T5"]},
329     {"id": "T9", "name": "Recharge / resupply", "depends_on":
330         []},
331     {"id": "T10", "name": "Re-sweep false-positive cell", "depends_on":
332         ["T2"]},
333 ]
334
335 MARITIME_TASK_TYPES: List[Dict] = [
336     {"id": "T1", "name": "Patrol hex sector", "depends_on":
337         []},
338     {"id": "T2", "name": "Classify AIS contact", "depends_on":
339         ["T1"]},
340     {"id": "T3", "name": "Intercept suspicious vessel", "depends_on":
341         ["T2"]},
342     {"id": "T4", "name": "Board & inspect vessel", "depends_on":
343         ["T3"]},

```



```

330     {"id": "T5", "name": "Escort vessel to port", "depends_on":
331         ["T4"]},
332     {"id": "T6", "name": "Update sector threat level", "depends_on":
333         ["T2"]},
334     {"id": "T7", "name": "Close coverage gap", "depends_on":
335         []},
336     {"id": "T8", "name": "Refuel / return to base", "depends_on":
337         []},
338     {"id": "T9", "name": "Maintain exclusion zone", "depends_on":
339         []},
340     {"id": "T10", "name": "Coordinate with aircraft", "depends_on":
341         ["T6"]},
342 ]
343
344 SCENARIO_TASK_TYPES: Dict[str, List[Dict]] = {
345     'warehouse': WAREHOUSE_TASK_TYPES,
346     'search_and_rescue': SAR_TASK_TYPES,
347     'maritime_security': MARITIME_TASK_TYPES,
348 }
349
350 # Export / sim paths
351 RESULTS_DIR = 'results'
352 MODELS_DIR = 'models'
353 SIM_READY_DIR = 'models/simulation_ready'
354 LOGS_DIR = 'logs'
355
356 # Sim-ready naming (matches inference_node.py regex)
357 # RL: sim_{ALGO}_{scenario}.pt
358 # Diffusion: sim_diff_{DIFF}_{RL}_{scenario}.pt
359 SIM_RL_PATTERN = 'sim_{algo}_{scenario}.pt'
360 SIM_DIFF_PATTERN = 'sim_diff_{diff}_{rl}_{scenario}.pt'
361
362 # DCMRTA improvements #21-24: Comprehensive metric keys
363 # From DCMRTA driver.py writeToTensorBoard:

```

```

358 # reward, valueLoss, policyLoss, entropy, gradNorm,
359 # success_rate, makespan, time_cost, waiting_time, travel_dist, efficiency
360 METRIC_KEYS = [
361     'reward', 'policy_loss', 'value_loss', 'entropy', 'grad_norm',
362     'completion_rate',      # fraction of tasks completed per episode
363     'makespan',             # steps to complete ALL tasks (lower = better)
364     'time_cost',            # mean time per task completion
365     'waiting_time',         # mean robot idle/waiting time
366     'travel_dist',          # total travel distance per episode
367     'efficiency',           # tasks_completed / travel_dist
368     'collisions',           # collision count per episode
369     'energy_consumed',      # total battery drain per episode
370     'reallocation_count',   # task reassignment count (deadlock recovery)
371 ]
372
373 # TA-RWARE: clash resolution (warehouse.py)
374 FIXING_CLASH_TIME = 4      # steps to wait before replanning after a clash
375 STUCK_THRESHOLD = 5        # steps at same position before deadlock assumed
376 # column_height-aware: actual threshold = STUCK_THRESHOLD + column_height
377                               + 2
378 COLUMN_HEIGHT = 3          # warehouse shelf column height (default 3 rows)
379 # Improvement K: position-aware stuck threshold
380 STUCK_THRESHOLD_REPLAN = STUCK_THRESHOLD                                #
381                               replan A* path
382 STUCK_THRESHOLD_FREE = STUCK_THRESHOLD + COLUMN_HEIGHT + 2              #
383                               free agent entirely
384
385 # DCMRTA improvement B: coalition patience threshold
386 # If a coalition partner arrives > max_waiting_time steps late, the
387     waiting agent abandons
388 COALITION_MAX_WAITING_TIME = 10    # matches DCMRTA
389     task_env.max_waiting_time = 10
390
391 # DCMRTA improvement A: normalised agent velocity

```

```

387 # Agents move at velocity=0.2 in a normalised  $[0,1]^2$  grid
388 # In real coordinates: normalised_velocity = max_speed / (2 * grid_half)
389 # Warehouse:  $1.5 / (2*25) = 0.030$  (grid 50x50, grid_half=25)
390 # SAR:  $2.5 / (2*75) = 0.017$  (grid 150x150, grid_half=75)
391 # Maritime:  $6.0 / (2*250) = 0.012$  (grid 500x500, grid_half=250)
392 NORMALISED_VELOCITY = {
393     'warehouse':      0.030,
394     'search_and_rescue': 0.017,
395     'maritime_security': 0.012,
396 }
397
398 # DCMRTA improvement G: confirmed observation feature dimensions
399 # From DCMRTA parameters.py: AGENT_INPUT_DIM=6, TASK_INPUT_DIM=5
400 # Agent features: [travel_time, remaining_work, waiting_time, rel_loc_x,
401                   rel_loc_y, assigned]
402 DCMRTA_AGENT_FEAT_DIM = 6
403 # Task features: [status, requirements, time, rel_loc_x, rel_loc_y]
404 DCMRTA_TASK_FEAT_DIM = 5
405
406 # DCMRTA improvement F: IL decay schedule
407 #  $il\_rate(ep) = base\_rate * \exp(IL\_DECAY * ep)$ 
408 # At  $ep=70000$ :  $0.5 * \exp(-1e-5 * 70000) = 0.5 * \exp(-0.7) \approx 0.25$ 
409 # At  $ep=200000$ :  $0.5 * \exp(-2.0) \approx 0.068$  (nearly off)
410 IL_DECAY_EXPONENT = -1e-5 # DCMRTA: negative exponent in  $\exp(coeff*ep)$ ;
411                             decay 0.5→0.25 over 70k episodes
412 DEMON_RATE = 0.5 # DCMRTA: initial IL mixing fraction
413
414 # DCMRTA: periodic evaluation settings
415 EVAL_INTERVAL_EPISODES = 500 # evaluate every 500 episodes
416 EVAL_N_SEEDS = 64 # 64 seeds per evaluation (fast)
417 EVAL_FULL_N_SEEDS = 256 # full 256 for final comparison
418 EVAL_P_THRESHOLD = 0.05 # paired t-test significance threshold
419
420 # DCMRTA: LR decay schedule (StepLR)

```

```

419 LR_DECAY_STEP = 2000    # decay every 2000 episodes
420 LR_DECAY_GAMMA = 0.98   # gentle decay:  $0.98^{10} = 0.817$  after 10 intervals
421
422 # Training MODES
423 # Single place to tune time budgets. run.py picks one of these via --mode.
424 #
425 # Mode intent:
426 # sanity    --- check all functions work as intended ( $\leq 10$  min)
427 # quick     --- confirm code runs end-to-end with no errors ( $\leq 10$  min)
428 # mini      --- confirm models are actually learning ( $\leq 1$  hour on GPU)
429 # 2hr       --- meaningful learning curves in ~2 hours
430 # full_train --- fully deployable model that works in sim (~16-20 h)
431 # full      --- alias for full_train (kept for backward compat)
432 # real_world --- online fine-tuning on physical robots (routed
    separately)
433 #
434 # Sizing (NVIDIA RTX A6000, 48 GB, 3-4 parallel workers):
435 # 15 RL jobs (A+B stages) + 15 diff jobs (C+D stages) = 30 total
436 MODES: Dict[str, Dict] = {
437     'sanity': {
438         #  $\leq 10$  min --- verify every code path executes without errors.
439         # 3 episodes, 1 worker, no AMP. Nothing is expected to learn;
440         # goal is "no crash, all imports resolve, all forward passes run".
441         'rl_episodes':      3,
442         'diff_epochs':      2,
443         'diff_collect':     4,
444         'max_steps':        50,
445         'num_workers':      1,
446         'rl_batch_size':    32,
447         'diff_batch_size':  16,
448         'ddim_steps':       5,
449         'use_amp':          False,
450         'async_pipeline':   False,
451         'use_hybrid_distill': False,

```

```

452         'log_every':          1,
453     },
454     'quick': {
455         # ≤10 min --- confirm code runs end-to-end with no errors.
456         # More episodes than sanity exercises more code-path variance.
457         'rl_episodes':        10,
458         'diff_epochs':        3,
459         'diff_collect':       8,
460         'max_steps':          100,
461         'num_workers':        2,
462         'rl_batch_size':      64,
463         'diff_batch_size':    32,
464         'ddim_steps':         5,
465         'use_amp':            False,
466         'async_pipeline':     False,
467         'use_hybrid_distill': False,
468         'log_every':          2,
469     },
470     'mini': {
471         # ≤1 hour --- confirm models are actually learning.
472         # 300 episodes is the minimum for a reliable reward-improvement
473         trend.
474         # max_steps=200 keeps per-episode time short.
475         # RL: 15 jobs x 300 ep x 200 steps ~ = 20-40 min on GPU + 4
476         workers.
477         # Diff: 15 jobs x 30 ep x 20-ep collect ~ = 10-15 min on GPU.
478         'rl_episodes':        300,
479         'diff_epochs':        30,
480         'diff_collect':       20,
481         'max_steps':          200,
482         'num_workers':        4,
483         'rl_batch_size':      256,
484         'diff_batch_size':    96,
485         'ddim_steps':         10,

```

```

484         'use_amp':                True,
485         'async_pipeline':         False,
486         'use_hybrid_distill':     True,
487         'log_every':              50,
488     },
489     '2hr': {
490         # ~2-hour compressed run. Meaningful learning curves, not
491         publication quality.
492         # RL: 15 jobs x 300 ep x 1200 steps ~= 60-70 min on 4 workers.
493         # Diff: 15 jobs x 15 ep x 30-ep collect ~= 25-35 min on 4 workers.
494         'rl_episodes':            300,
495         'diff_epochs':            15,
496         'diff_collect':           30,
497         'max_steps':              1200,
498         'num_workers':            4,
499         'rl_batch_size':          512,
500         'diff_batch_size':        256,
501         'ddim_steps':             20,
502         'use_amp':                True,
503         'async_pipeline':         False,
504         'use_hybrid_distill':     True,
505         'log_every':              50,
506     },
507     'full_train': {
508         # ~16-20 hour run targeting a fully deployable model for sim.
509         # 5000 RL episodes + 300 diff epochs target convergence, not just
510         a
511         # learning trend. Use --rl_episodes 8000 for publication-quality
512         runs.
513         # RL: 15 jobs x 5000 ep x 1200 steps ~= 14-18 h on 3 workers.
514         # Diff: 15 jobs x 300 ep x 200-ep collect ~= 4-6 h on 3 workers.
515         'rl_episodes':            5000,
516         'diff_epochs':            300,
517         'diff_collect':           200,

```

```

515     'max_steps':          1200,
516     'num_workers':        3,
517     'rl_batch_size':      512,
518     'diff_batch_size':    256,
519     'ddim_steps':         20,
520     'use_amp':            True,
521     'async_pipeline':     False,
522     'use_hybrid_distill': True,
523     'log_every':          200,
524 },
525 'full': {
526     # Kept as backward-compatible alias for full_train.
527     # RL: 15 jobs x 3000 ep x 1200 steps ~= 14-16 h on 3 workers.
528     # Diff: 15 jobs x 200 ep x 150-ep collect ~= 4-6 h on 3 workers.
529     'rl_episodes':        3000,
530     'diff_epochs':        200,
531     'diff_collect':       150,
532     'max_steps':          1200,
533     'num_workers':        3,
534     'rl_batch_size':      512,
535     'diff_batch_size':    256,
536     'ddim_steps':         20,
537     'use_amp':            True,
538     'async_pipeline':     False,
539     'use_hybrid_distill': True,
540     'log_every':          200,
541 },
542 'real_world': {
543     # Online fine-tuning deployed on physical robots.
544     # run.py routes this mode directly to _run_real_world(); the
staged
545     # pipeline is not used. Parameters here configure the
OnlineTrainer.
546     # 1 worker (the robot itself), no hybrid distillation, small

```

```

    batches
547     # for low-latency gradient updates.
548     'rl_episodes':      500,
549     'diff_epochs':      5,
550     'diff_collect':     10,
551     'max_steps':        200,
552     'num_workers':      1,
553     'rl_batch_size':    64,
554     'diff_batch_size':  32,
555     'ddim_steps':       10,
556     'use_amp':          False,
557     'async_pipeline':   False,
558     'use_hybrid_distill': False,
559     'log_every':        10,
560 },
561 }
562
563
564 def apply_mode(mode: str) -> Dict:
565     """Return the mode dict and mutate
566     RLConfig/DiffusionConfig/ParallelConfig
567     defaults in-place so workers built post-fork pick up the right values.
568
569     Modes: sanity (<=10 min code check), quick (<=10 min no-error run),
570     mini (<=1 hr learning check), 2hr, full_train (~16-20 hr deployable),
571     full (alias for full_train), real_world (online robot fine-tuning).
572
573     Environment overrides applied after mode defaults (so they win):
574     AQUA_RL_BATCH_SIZE    --- override RLConfig.batch_size
575     AQUA_DIFF_BATCH_SIZE  --- override DiffusionConfig.batch_size
576
577     A malformed env-var value raises ValueError (no silent fallback).
578     """
579     import os as _os

```



```

579     if mode not in MODES:
580         raise ValueError(f'Unknown mode: {mode!r}. Valid: {list(MODES)}')
581     m = MODES[mode]
582
583     # RL config
584     RLConfig.num_episodes = m['rl_episodes']
585     RLConfig.max_steps    = m['max_steps']
586     RLConfig.batch_size   = m['rl_batch_size']
587
588     # Diffusion config
589     DiffusionConfig.num_epochs = m['diff_epochs']
590     DiffusionConfig.batch_size = m['diff_batch_size']
591     DiffusionConfig.ddim_steps = m['ddim_steps']
592
593     # Parallel config
594     ParallelConfig.rl_workers   = m['num_workers']
595     ParallelConfig.diff_workers = m['num_workers']
596
597     # Env-var overrides (set by run.py or manually). No silent try/except:
598     # a malformed value raises and surfaces the cause.
599     rl_ov = _os.environ.get('AQUA_RL_BATCH_SIZE', '').strip()
600     if rl_ov:
601         RLConfig.batch_size = max(16, int(rl_ov))
602     diff_ov = _os.environ.get('AQUA_DIFF_BATCH_SIZE', '').strip()
603     if diff_ov:
604         DiffusionConfig.batch_size = max(8, int(diff_ov))
605
606     return m

```

Listing B.2: Central configuration: dimensions, mode presets, RL and diffusion hyperparameters, scenario list and path constants.

Package: environment

```
1 """
2 environment/__init__.py
3 =====
4 All environment components - fully self-contained, no external file
5     dependencies.
6
7 Files:
8     task_env.py - ImprovedTaskAllocationEnv and supporting classes
9     env_config.py - DatasetConfig, RLConfig, global config instances
10    observation_extractors.py - Obs extractors, Kalman predictor, collision
11        checks
12    reward_wrapper.py - RewardShapingWrapper with potential shaping +
13        exploration
14    reward_shaping.py - Scenario-aware reward shaper (TA-RWARE style)
15    analysis.py - EDA + results export helpers
16
17 note: training_loop.py / fallback_train_rl removed. It was dead code left
18     over
19 from an older agent interface (select_actions / step=...) that did not
20     match the
21 current models/rl_agents.py. Every RL job now routes through
22 training.rl_trainer.train_rl_agent directly.
23
24 note: the previous try/except around the import block was a silent
25     fallback
26 (_ENV_AVAILABLE = False on error) that hid real bugs. Removed per the
27 "no fallbacks" policy: an import error now surfaces immediately.
28 """
29 from environment.task_env import ImprovedTaskAllocationEnv,
30     CurriculumManager
31 from environment.env_config import DatasetConfig, RLConfig as EnvRLConfig
32 from environment.env_config import dataset_config, rl_config
```

```

26 from environment.reward_wrapper import (
27     RewardShapingWrapper, SCENARIO_CFG, make_curriculum, build_env)
28
29 __all__ = [
30     'ImprovedTaskAllocationEnv', 'CurriculumManager',
31     'DatasetConfig', 'EnvRLConfig', 'dataset_config', 'rl_config',
32     'RewardShapingWrapper', 'SCENARIO_CFG', 'make_curriculum',
33     'build_env',
34 ]

```

Listing B.3: Package init for the environment package.

```

1 """
2 environment/env_config.py
3 =====
4 DatasetConfig and RLConfig singletons used by ImprovedTaskAllocationEnv.
5
6 Grid sizes align with config/base_config.py SCENARIO_CONFIGS:
7     warehouse 50x50 m (grid_half=25), SAR 100x100 m (grid_half=50),
8     maritime 500x500 m (grid_half=250).
9
10 Obstacle counts are density-normalised to the grid area so that obstacle
11 density stays constant regardless of grid size. Physical radii (robot 0.4
12     /
13     0.65 / 3.0 m, shelf 0.5-1.5 m, vessel 3 m) are real-world values and
14     fixed.
15
16 validate() raises AssertionError immediately on misconfiguration; no
17     silent
18     fallbacks.
19 """
20
21 from __future__ import annotations
22 import logging
23 from dataclasses import dataclass, field

```

```

20 from typing import List, Dict, Tuple
21
22 logger = logging.getLogger(__name__)
23
24
25 @dataclass
26 class DatasetConfig:
27     """Configuration for dataset generation with scenario-specific
28         parameters."""
29
30     # Dataset sizes
31     SMALL_DATASET_SIZE_MB: int = 100
32     LARGE_DATASET_SIZE_MB: int = 15000
33     ESTIMATED_SAMPLES_PER_MB: int = 20 # For progress tracking
34
35     # Scenarios
36     SCENARIOS: List[str] = field(default_factory=lambda: [
37         'warehouse', 'search_and_rescue', 'maritime_security'])
38
39     # Algorithm lists - COMPLETE SET
40     RL_ALGORITHMS: List[str] = field(default_factory=lambda: [
41         'MAPPO', 'QMIX', 'MADDPG', 'DQN', 'HybridRL'])
42     DIFFUSION_ALGORITHMS: List[str] = field(default_factory=lambda: [
43         'TDM', 'DiffusionPolicy', 'Diffuser', 'SDE', 'HybridDiffusion'])
44
45     #
46     # Warehouse scenario
47     # grid 20x20 -> 14x14 (area factor 0.49).
48     # Shelves: 50 -> 25 (keep density ~0.125 shelves/m^2 across both
49     # sizes).
50     #
51     WAREHOUSE_GRID_SIZE: Tuple[int, int] = (50, 50)
52     WAREHOUSE_NUM_SHELVES: int = 80
53     WAREHOUSE_NUM_STATIONS: int = 8

```

```

52 WAREHOUSE_OBSTACLE_DENSITY: float = 0.3
53 WAREHOUSE_TERRAIN_ROUGHNESS: float = 1.0 # For battery drain
54
55 #
56 # Search and rescue scenario
57 # grid 50x50 -> 30x30 (area factor 0.36).
58 # Obstacles: 100 -> 36 (keep density ~0.04 obstacles/m^2).
59 # Targets and hazard zones unchanged (they are scenario entities, not
60 # obstacles - their density is already set by num_tasks in
61 base_config).
62
63 SAR_GRID_SIZE: Tuple[int, int] = (100, 100)
64 SAR_NUM_OBSTACLES: int = 44 # scaled from 100 proportionally to new
65 grid area
66 SAR_NUM_TARGETS: int = 50
67 SAR_OBSTACLE_DENSITY: float = 0.2
68 SAR_HAZARD_ZONES: int = 5
69 SAR_TERRAIN_ROUGHNESS: float = 2.5 # Rougher terrain
70
71 #
72 # Maritime security scenario
73 # v4b: grid 200x200 -> 80x80 (area ratio 0.444 vs 120x120).
74 # Object counts scaled by (80/120)^2 = 0.444:
75 # Vessels: 5 -> 3 | Buoys: 11 -> 5
76 # Patrol zones: 3 -> 2 | Restricted areas: 2 -> 1
77 # Physical radii unchanged (real-world m); restricted-area exclusion
78 zones
79 # scaled proportionally: 8-20 m -> 5-13 m.
80
81 #
82 MARITIME_GRID_SIZE: Tuple[int, int] = (500, 500)
83 MARITIME_NUM_VESSELS: int = 8
84 MARITIME_NUM_BUOYS: int = 15
85 MARITIME_NUM_PATROL_ZONES: int = 5
86 MARITIME_NUM_RESTRICTED_AREAS: int = 3

```

```

83 MARITIME_CURRENT_STRENGTH_RANGE: Tuple[float, float] = (0.5, 2.0)
84 MARITIME_WAVE_HEIGHT_RANGE: Tuple[float, float] = (0.5, 3.0)
85 MARITIME_WIND_SPEED_RANGE: Tuple[float, float] = (5.0, 20.0)
86 MARITIME_TERRAIN_ROUGHNESS: float = 1.5 # Water resistance
87
88 #
89 # Global robot parameter defaults (used as fallback for unknown
scenarios)
90 #
91 NUM_ROBOTS_RANGE: Tuple[int, int] = (2, 20)
92 ROBOT_RADIUS: float = 0.5
93 ROBOT_MAX_SPEED: float = 2.0
94 ROBOT_MAX_SPEED_LOADED: float = 1.0
95 ROBOT_MAX_PAYLOAD: float = 50.0 # kg
96 ROBOT_WEIGHT: float = 30.0 # kg
97 ROBOT_BATTERY_CAPACITY: float = 100.0 # Wh
98 ROBOT_BATTERY_DRAIN_RATE: float = 0.1 # Base drain rate (Wh/s)
99 ROBOT_BATTERY_DRAIN_LOADED_FACTOR: float = 1.5
100 ROBOT_BATTERY_DRAIN_SPEED_COEF: float = 0.02 # Speed2 coefficient
101 ROBOT_OBSERVATION_RADIUS: float = 10.0
102 ROBOT_COMMUNICATION_RANGE: float = 15.0
103 ROBOT_REGEN BRAKING EFFICIENCY: float = 0.3
104
105 #
106 # Warehouse: compact indoor AMRs (Autonomous Mobile Robots)
107 # obs/comm radii reduced to match smaller 14x14 grid.
108 # A 12 m comm range in a 14 m grid  $\approx$  covers the whole floor, which is
109 # realistic for a warehouse (typical Wi-Fi / UWB mesh), but we drop it
110 # slightly so nearby-robot encoders actually have to filter.
111 #
112 WAREHOUSE_ROBOT_PARAMS: Dict = field(default_factory=lambda: {
113     'num_robots_range': (2, 20),
114     'radius': 0.4, # Narrow aisles (physical)
115     'max_speed': 1.5, # Indoor speed limit (m/s)

```

```

116         'max_speed_loaded': 0.75,
117         'max_payload': 80.0,                # Heavy package handling (kg)
118         'weight': 25.0,
119         'battery_capacity': 80.0,
120         'battery_drain_rate': 0.07,
121         'battery_drain_loaded_factor': 1.4,
122         'battery_drain_speed_coef': 0.012,
123         'observation_radius': 10.0,
124         'communication_range': 20.0,
125         'regen_braking_efficiency': 0.40,
126     })
127
128     #
129     # Search & Rescue: rugged field robots for rough/outdoor terrain
130     # obs/comm radii scaled 5030.
131     #
132     SAR_ROBOT_PARAMS: Dict = field(default_factory=lambda: {
133         'num_robots_range': (2, 20),
134         'radius': 0.65,                    # Larger for uneven terrain
135         'max_speed': 2.5,
136         'max_speed_loaded': 1.0,
137         'max_payload': 30.0,
138         'weight': 45.0,
139         'battery_capacity': 160.0,
140         'battery_drain_rate': 0.20,
141         'battery_drain_loaded_factor': 1.9,
142         'battery_drain_speed_coef': 0.038,
143         'observation_radius': 25.0,
144         'communication_range': 50.0,
145         'regen_braking_efficiency': 0.12,
146     })
147
148     #
149     # Maritime Security: autonomous surface vessels / USVs

```

```

150     # obs/comm radii scaled 200120.
151     # Vessel radius 3 m is physical and kept; it is now a slightly larger
152     # fraction of the 120 m grid but still reasonable (2.5 %).
153     #
154     MARITIME_ROBOT_PARAMS: Dict = field(default_factory=lambda: {
155         'num_robots_range': (2, 20),
156         'radius': 3.0,                                # Large vessel footprint
157                                                         (physical)
158         'max_speed': 6.0,                                # ~12 knots (m/s)
159         'max_speed_loaded': 3.5,
160         'max_payload': 500.0,
161         'weight': 800.0,
162         'battery_capacity': 600.0,
163         'battery_drain_rate': 0.55,
164         'battery_drain_loaded_factor': 1.7,
165         'battery_drain_speed_coef': 0.090,
166         'observation_radius': 100.0,
167         'communication_range': 200.0,
168         'regen_braking_efficiency': 0.08,
169     })
170
171     def get_robot_params(self, scenario: str) -> Dict:
172         """Return scenario-specific robot parameters dict."""
173         if scenario == 'warehouse':
174             return self.WAREHOUSE_ROBOT_PARAMS
175         elif scenario == 'search_and_rescue':
176             return self.SAR_ROBOT_PARAMS
177         elif scenario == 'maritime_security':
178             return self.MARITIME_ROBOT_PARAMS
179         else:
180             # Build a dict from the global flat params as a generic
181             # default.
182             # This is NOT a silent fallback - it is the documented
183             # behaviour

```



```

181         # for unknown scenario names and callers opt into it by
182         passing
183         # a string that doesn't match the three named scenarios.
184         return {
185             'num_robots_range': self.NUM_ROBOTS_RANGE,
186             'radius': self.ROBOT_RADIUS,
187             'max_speed': self.ROBOT_MAX_SPEED,
188             'max_speed_loaded': self.ROBOT_MAX_SPEED_LOADED,
189             'max_payload': self.ROBOT_MAX_PAYLOAD,
190             'weight': self.ROBOT_WEIGHT,
191             'battery_capacity': self.ROBOT_BATTERY_CAPACITY,
192             'battery_drain_rate': self.ROBOT_BATTERY_DRAIN_RATE,
193             'battery_drain_loaded_factor':
194                 self.ROBOT_BATTERY_DRAIN_LOADED_FACTOR,
195             'battery_drain_speed_coef':
196                 self.ROBOT_BATTERY_DRAIN_SPEED_COEF,
197             'observation_radius': self.ROBOT_OBSERVATION_RADIUS,
198             'communication_range': self.ROBOT_COMMUNICATION_RANGE,
199             'regen_braking_efficiency':
200                 self.ROBOT_REGEN_BRAKING_EFFICIENCY,
201         }
202
203     # Trajectory parameters
204     TRAJECTORY_HORIZON: int = 64 # Must be divisible by 4
205     TRAJECTORY_DURATION: float = 5.0
206
207     # Task parameters
208     TASK_PRIORITIES: List[str] = field(default_factory=lambda: ['high',
209         'medium', 'low'])
210     TASK_TYPES_WAREHOUSE: List[str] = field(default_factory=lambda: [
211         'pick', 'place', 'transport', 'sort'])
212     TASK_TYPES_SAR: List[str] = field(default_factory=lambda: [
213         'search', 'rescue', 'scan', 'deliver'])
214     TASK_TYPES_MARITIME: List[str] = field(default_factory=lambda: [

```

```

210         'patrol', 'intercept', 'surveillance', 'escort',
211         'investigate', 'perimeter_defense'])
212
213     # Task dependency parameters
214     MAX_TASK_DEPENDENCIES: int = 3
215     DEPENDENCY_PROBABILITY: float = 0.3
216     MIN_TASK_SEPARATION_TIME: float = 1.0
217
218     # Scenario-specific task generation parameters
219     # Warehouse: small, frequent packages on tight deadlines
220     WAREHOUSE_TASK_PARAMS: Dict = field(default_factory=lambda: {
221         'payload_weight_range': (1.0, 50.0),      # kg - packages up to 50
222         kg
223         'deadline_range': (20.0, 60.0),          # steps - tight indoor
224         schedule
225         'duration_range': (0.5, 5.0),            # sim-seconds; DT=0.078
226         → 6-64 steps
227         'priority_probs': [0.35, 0.45, 0.20],    # high / medium / low
228         'dependency_prob': 0.15,                # Sequential
229         pick-then-place
230     })
231
232     # SAR: diverse objectives - light scan to heavy rescue carry
233     SAR_TASK_PARAMS: Dict = field(default_factory=lambda: {
234         'payload_weight_range': (0.0, 25.0),      # kg - injured person or
235         equipment
236         'deadline_range': (50.0, 200.0),          # steps - long outdoor
237         missions
238         'duration_range': (1.0, 8.0),            # sim-seconds; DT=0.078
239         → 13-102 steps
240         'priority_probs': [0.50, 0.35, 0.15],    # High urgency overall
241         'dependency_prob': 0.25,                # Scan then rescue
242         sequence
243     })
244
245     # Maritime: large cargo, long patrol zones, relaxed deadlines

```

```

236 MARITIME_TASK_PARAMS: Dict = field(default_factory=lambda: {
237     'payload_weight_range': (50.0, 450.0),    # kg - seized goods or
        equipment
238     'deadline_range': (100.0, 500.0),        # steps - long ocean
        patrols
239     'duration_range': (1.0, 6.0),            # sim-seconds; DT=0.078
        → 13-77 steps
240     'priority_probs': [0.25, 0.45, 0.30],    # Many routine patrols
241     'dependency_prob': 0.10,                 # More independent tasks
        at sea
242 })
243
244 def get_task_params(self, scenario: str) -> Dict:
245     """Return scenario-specific task generation parameters."""
246     if scenario == 'warehouse':
247         return self.WAREHOUSE_TASK_PARAMS
248     elif scenario == 'search_and_rescue':
249         return self.SAR_TASK_PARAMS
250     elif scenario == 'maritime_security':
251         return self.MARITIME_TASK_PARAMS
252     else:
253         return {
254             'payload_weight_range': (5.0, 30.0),
255             'deadline_range': (30.0, 100.0),
256             'duration_range': (5.0, 20.0),
257             'priority_probs': [0.30, 0.50, 0.20],
258             'dependency_prob': 0.20,
259         }
260
261     # Synthetic data parameters
262     WEIGHT_VARIATION: float = 0.2
263     SPEED_LOADED_FACTOR: float = 0.6
264
265     def get_normalization_factors(self, scenario: str) -> Dict[str,

```

```

float]:
266     """Get scenario-specific normalization factors."""
267     if scenario == 'warehouse':
268         grid_size = max(self.WAREHOUSE_GRID_SIZE)
269     elif scenario == 'search_and_rescue':
270         grid_size = max(self.SAR_GRID_SIZE)
271     elif scenario == 'maritime_security':
272         grid_size = max(self.MARITIME_GRID_SIZE)
273     else:
274         grid_size = 30.0 # default changed from 50 to 30 to match
new SAR
275
276     return {
277         'position': grid_size,
278         'velocity': self.get_robot_params(scenario)['max_speed'],
279         'battery':
self.get_robot_params(scenario)['battery_capacity'],
280         'payload': self.get_robot_params(scenario)['max_payload'],
281         'time': self.TRAJECTORY_DURATION
282     }
283
284     @property
285     def DT(self) -> float:
286         """Timestep duration."""
287         return self.TRAJECTORY_DURATION / self.TRAJECTORY_HORIZON
288
289     def validate(self):
290         """Validate configuration parameters. Raises AssertionError on
bad config."""
291         assert self.TRAJECTORY_HORIZON % 4 == 0, "TRAJECTORY_HORIZON must
be divisible by 4"
292         assert self.NUM_ROBOTS_RANGE[0] >= 1, "Must have at least 1 robot"
293         assert 0 < self.WEIGHT_VARIATION < 1, "Weight variation must be
between 0 and 1"

```

```

294         for sc in self.SCENARIOS:
295             p = self.get_robot_params(sc)
296             assert p['max_speed'] > p['max_speed_loaded'], \
297                 f"{sc}: loaded speed must be < unloaded speed"
298             # extra: sanity check grid sizes match base_config's grid_half *
299             2.
300             expected_grid = {
301                 'warehouse': 50,      # grid_half = 25
302                 'search_and_rescue': 100,  # grid_half = 50
303                 'maritime_security': 500,  # grid_half = 250
304             }
305             actual_grid = {
306                 'warehouse': max(self.WAREHOUSE_GRID_SIZE),
307                 'search_and_rescue': max(self.SAR_GRID_SIZE),
308                 'maritime_security': max(self.MARITIME_GRID_SIZE),
309             }
310             for sc, exp in expected_grid.items():
311                 act = actual_grid[sc]
312                 assert act == exp, (
313                     f"{sc}: env_config grid {act} disagrees with base_config
314                     expected {exp}. "
315                     "env_config.py and config/base_config.py
316                     SCENARIO_CONFIGS.grid_half "
317                     "must agree.")
318                 logger.info("DatasetConfig validated successfully (grids 50 / 150
319                             / 500)")
320
321     @dataclass
322     class RLConfig:
323         """Configuration for RL algorithms with all improvements."""
324
325         RL_ALGORITHMS: List[str] = field(default_factory=lambda: [
326             'MAPPO', 'QMIX', 'MADDPG', 'DQN', 'HybridRL'])

```

```

324
325     # Common parameters
326     NUM_EPISODES: int = 5000
327     MAX_STEPS_PER_EPISODE: int = 200
328     BATCH_SIZE: int = 256
329     LEARNING_RATE: float = 3e-4
330     GAMMA: float = 0.99
331
332     # Network architecture
333     HIDDEN_DIM: int = 256
334     NUM_LAYERS: int = 3
335     USE_ATTENTION: bool = True
336     NUM_ATTENTION_HEADS: int = 4
337     DROPOUT_RATE: float = 0.1
338
339     # MAPPO specific
340     GAE_LAMBDA: float = 0.95
341     CLIP_EPSILON: float = 0.2
342     PPO_EPOCHS: int = 10
343     VALUE_LOSS_COEF: float = 0.5
344     ENTROPY_COEF: float = 0.01
345     MAX_GRAD_NORM: float = 0.5
346     MAPPO_ROLLOUT_LENGTH: int = 2048
347     MAPPO_USE_CENTRALIZED_CRITIC: bool = True
348     MAPPO_SHARE_ACTOR_PARAMS: bool = True
349     MAPPO_NORMALIZE_ADVANTAGES: bool = True
350
351     # QMIX specific
352     MIXING_NETWORK_DIM: int = 64
353     HYPERNET_DIM: int = 128
354     TARGET_UPDATE_FREQUENCY: int = 200
355     QMIX_MONOTONIC_CONSTRAINT: bool = True
356     QMIX_HYPERNETWORK_GRAD_CLIP: float = 10.0
357

```

```

358     # MADDPG specific (discrete-action Gumbel-softmax variant - see
        models/rl_agents.py)
359     TAU: float = 0.01
360     ACTOR_LR: float = 1e-4
361     CRITIC_LR: float = 1e-3
362     NOISE_SCALE: float = 0.1
363     NOISE_DECAY: float = 0.9999
364     NOISE_MIN: float = 0.01
365     # Gumbel-softmax temperature for discrete MADDPG
366     MADDPG_GUMBEL_TEMP_START: float = 1.0
367     MADDPG_GUMBEL_TEMP_END: float = 0.1
368     MADDPG_GUMBEL_TEMP_DECAY: int = 5000 # episodes to decay over
369
370     # DQN specific (these flags are now actually honoured by
        rl_agents.DQN)
371     REPLAY_BUFFER_SIZE: int = 100000
372     EPSILON_START: float = 1.0
373     EPSILON_END: float = 0.01
374     EPSILON_DECAY: float = 0.995
375     USE_DOUBLE_DQN: bool = True
376     USE_DUELING_DQN: bool = True
377     USE_PER: bool = True
378     PER_ALPHA: float = 0.6
379     PER_BETA_START: float = 0.4
380     PER_BETA_END: float = 1.0
381     PER_EPS: float = 1e-6
382     PER_MIN_BUFFER_SIZE: int = 128
383
384     # Hybrid RL specific
385     HYBRID_VOTING_WEIGHTS: Dict[str, float] =
        field(default_factory=lambda: {
386         'MAPPO': 0.3, 'QMIX': 0.25, 'MADDPG': 0.25, 'DQN': 0.2
387     })
388     ADAPTIVE_WEIGHTS: bool = True

```

```

389     HYBRID_USE_UCB: bool = True
390     UCB_C: float = 2.0
391
392     # Environment settings
393     MAX_ROBOTS: int = 20
394     MAX_TASKS: int = 50
395     MAX_OBSTACLES: int = 100
396
397     # Reward weights
398     REWARD_TASK_COMPLETION: float = 20.0
399     REWARD_PRIORITY_BONUS: float = 5.0
400     PENALTY_DELAY: float = -1.0
401     PENALTY_OVERLOAD: float = -5.0
402     PENALTY_INCOMPLETE: float = -0.3
403     REWARD_EFFICIENCY: float = 2.0
404     REWARD_COLLISION_FREE: float = 1.0
405     PENALTY_COLLISION: float = -0.05
406     PENALTY_ROBOT_COLLISION: float = -2.0
407     REWARD_COORDINATION_BONUS: float = 3.0
408     USE_DIFFERENCE_REWARDS: bool = True
409
410     # Curriculum learning
411     USE_CURRICULUM: bool = False
412     CURRICULUM_LEVELS: int = 5
413     CURRICULUM_THRESHOLD: float = 0.7
414     CURRICULUM_WINDOW: int = 100
415
416     # Training robustness
417     MAX_RETRIES_ON_ERROR: int = 3
418     REDUCE_LR_ON_ERROR: float = 0.5
419     NAN_DETECTION: bool = True
420     MIN_TEMPERATURE: float = 1e-8
421     SAVE_RNG_STATES: bool = True
422

```



```

423     # Evaluation
424     EVAL_FREQUENCY: int = 100
425     NUM_EVAL_EPISODES: int = 10
426     SAVE_FREQUENCY: int = 500
427
428     # Learning rate scheduling
429     USE_LR_SCHEDULER: bool = True
430     LR_SCHEDULER_TYPE: str = 'cosine'
431     LR_WARMUP_STEPS: int = 1000
432
433     # Paths
434     LOG_DIR: str = './logs'
435     CHECKPOINT_DIR: str = './checkpoints'
436     RESULTS_DIR: str = './results'
437
438     def validate(self):
439         """Validate RL configuration. Raises AssertionError on bad
440         config."""
441         assert self.BATCH_SIZE > 0
442         assert 0 < self.GAMMA <= 1
443         assert self.GAE_LAMBDA >= 0
444         logger.info("RLConfig validated successfully")
445
446     #
447     # Global config instances (used by ImprovedTaskAllocationEnv)
448     # no silent try/except around validation - bad config surfaces
449     #
450     dataset_config = DatasetConfig()
451     rl_config      = RLConfig()
452     dataset_config.validate()
453     rl_config.validate()
454

```

```

455
456 #
457 # DCMRTA: reactive planning (env/task_env.py)
458 # When True, tasks are revealed progressively (maritime: 20 40... visible)
459 # rather than all at once. Set per-scenario; particularly useful for
      maritime.
460 #
461 REACTIVE_PLANNING_SCENARIOS = {
462     'warehouse':          False,    # all shelves visible
463     'search_and_rescue': False,    # all victim locations known
464     'maritime_security': True,     # threats appear progressively
465 }
466 REACTIVE_VISIBILITY_STEP = 20      # reveal 20 new tasks every 10 time units

```

Listing B.4: Per-scenario environment constants: grid sizes, robot counts, communication ranges, battery dynamics and reward coefficients.

```

1  """
2  environment/observation_extractors.py
3  =====
4  Observation extraction classes for AQUA Guardian.
5
6  Builds the three observation tensors (local, global, flat) from raw
7  environment state. Scenario-specific normalisation constants come from
8  config.base_config.SCENARIO_NORM so all three obs types stay consistent
9  with the environment's coordinate frame.
10 """
11 import os
12 import sys
13 import gc
14 import pickle
15 import time
16 import math
17 import heapq

```

```

18 import numpy as np
19 import pandas as pd
20 import torch
21 import torch.nn as nn
22 import torch.nn.functional as F
23 import torch.optim as optim
24 from torch.optim.lr_scheduler import CosineAnnealingLR, LinearLR,
    OneCycleLR
25 from torch.distributions import Categorical, Normal
26 try:
27     from torch.amp import autocast, GradScaler          # PyTorch >= 2.0
28 except ImportError:
29     from torch.cuda.amp import autocast, GradScaler     # PyTorch < 2.0
30 import matplotlib
31 matplotlib.use('Agg') # Non-interactive backend for headless servers
32 import matplotlib.pyplot as plt
33 from pathlib import Path
34 from typing import List, Dict, Tuple, Optional, Any, Union, Callable,
    NamedTuple
35 import json
36 from datetime import datetime
37 from collections import deque, defaultdict
38 from abc import ABC, abstractmethod
39 from dataclasses import dataclass, field
40 from scipy import stats
41 from scipy.optimize import linear_sum_assignment
42 import warnings
43 import copy
44 import logging
45 import traceback
46 import threading
47 from contextlib import contextmanager
48 from functools import lru_cache
49 import random

```

```

50
51 warnings.filterwarnings('ignore')
52
53 import gymnasium as gym
54 from gymnasium import spaces
55 from environment.env_config import (
56     DatasetConfig, RLConfig,
57     dataset_config, rl_config
58 )
59 from config.base_config import (
60     TASK_STATUS_UNAVAILABLE, TASK_STATUS_AVAILABLE,
61     TASK_STATUS_IN_PROGRESS, TASK_STATUS_COMPLETE,
62     ROBOT_STATUS_ACTIVE,
63 )
64
65 logger = logging.getLogger(__name__)
66
67
68
69 class TaskDependencyValidator:
70     """
71     Validate task dependencies to ensure:
72     1. Dependencies form a DAG (no cycles)
73     2. Dependent tasks have later deadlines than prerequisites
74     3. Dependency chains don't exceed max_depth
75     """
76
77     @staticmethod
78     def topological_sort(tasks: List[Dict]) -> Optional[List[int]]:
79         """
80         Perform topological sort to verify DAG
81
82         Args:
83         tasks: List of task dictionaries with 'id' and 'dependencies'

```

```

84
85 Returns:
86 Sorted task IDs if DAG is valid, None if cycle detected
87 """
88     # Build adjacency list and in-degree count
89     adj_list = defaultdict(list)
90     in_degree = defaultdict(int)
91     task_map = {task['id']: task for task in tasks}
92
93     for task in tasks:
94         task_id = task['id']
95         in_degree[task_id] = 0 # Initialize
96
97     for task in tasks:
98         task_id = task['id']
99         deps = task.get('dependencies', [])
100
101         for dep_id in deps:
102             if dep_id not in task_map:
103                 logger.warning(f"Task {task_id} has invalid
dependency {dep_id}")
104                 continue
105                 adj_list[dep_id].append(task_id)
106                 in_degree[task_id] += 1
107
108     # Kahn's algorithm
109     queue = deque([tid for tid in in_degree if in_degree[tid] == 0])
110     sorted_tasks = []
111
112     while queue:
113         task_id = queue.popleft()
114         sorted_tasks.append(task_id)
115
116         for neighbor in adj_list[task_id]:

```

```

117         in_degree[neighbor] -= 1
118         if in_degree[neighbor] == 0:
119             queue.append(neighbor)
120
121         # Check if all tasks are sorted (no cycles)
122         if len(sorted_tasks) != len(tasks):
123             logger.error("Cycle detected in task dependencies!")
124             return None
125
126         return sorted_tasks
127
128     @staticmethod
129     def validate_and_fix_deadlines(tasks: List[Dict], min_separation:
130         float = 1.0):
131         """
132         Adjust deadlines to ensure dependent tasks have later deadlines
133
134         Args:
135         tasks: List of task dictionaries
136         min_separation: Minimum time between dependent tasks
137         """
138
139         task_map = {task['id']: task for task in tasks}
140
141         # Get topological order
142         topo_order = TaskDependencyValidator.topological_sort(tasks)
143         if topo_order is None:
144             raise ValueError("Cannot fix deadlines: dependency cycle
145                 detected")
146
147         # Process in topological order
148         for task_id in topo_order:
149             task = task_map[task_id]
150             deps = task.get('dependencies', [])

```

```

149         if deps:
150             # Find maximum deadline of all dependencies
151             max_dep_deadline = max(
152                 task_map[dep_id]['deadline'] + min_separation
153                 for dep_id in deps
154                 if dep_id in task_map
155             )
156
157             # Ensure this task's deadline is after all dependencies
158             if task['deadline'] < max_dep_deadline:
159                 old_deadline = task['deadline']
160                 task['deadline'] = max_dep_deadline
161                 logger.debug(f"Adjusted task {task_id} deadline:
162 {old_deadline:.2f} → {task['deadline']:.2f}")
163
164     @staticmethod
165     def get_dependency_depth(tasks: List[Dict]) -> Dict[int, int]:
166         """Calculate depth of each task in dependency chain"""
167         task_map = {task['id']: task for task in tasks}
168         depths = {}
169
170         def calculate_depth(task_id: int) -> int:
171             if task_id in depths:
172                 return depths[task_id]
173
174             task = task_map.get(task_id)
175             if not task:
176                 return 0
177
178             deps = task.get('dependencies', [])
179             if not deps:
180                 depths[task_id] = 0
181                 return 0

```

```

182         max_dep_depth = max(
183             calculate_depth(dep_id)
184             for dep_id in deps
185             if dep_id in task_map
186         )
187         depths[task_id] = max_dep_depth + 1
188         return depths[task_id]
189
190     for task in tasks:
191         calculate_depth(task['id'])
192
193     return depths
194
195     @staticmethod
196     def validate_tasks(tasks: List[Dict], max_depth: int = 5) -> bool:
197         """
198         Validate all task constraints
199
200         Returns:
201         True if valid, raises ValueError if invalid
202         """
203         # Check for DAG
204         if TaskDependencyValidator.topological_sort(tasks) is None:
205             raise ValueError("Task dependencies contain a cycle")
206
207         # Check dependency depth
208         depths = TaskDependencyValidator.get_dependency_depth(tasks)
209         max_actual_depth = max(depths.values()) if depths else 0
210
211         if max_actual_depth > max_depth:
212             raise ValueError(f"Dependency chain too deep:
213                               {max_actual_depth} > {max_depth}")
214
215         # Validate deadlines

```



```

215     task_map = {task['id']: task for task in tasks}
216     for task in tasks:
217         deps = task.get('dependencies', [])
218         for dep_id in deps:
219             if dep_id in task_map:
220                 if task['deadline'] <= task_map[dep_id]['deadline']:
221                     logger.warning(f"Task {task['id']} deadline not
after dependency {dep_id}")
222
223     logger.info(f" Tasks validated: {len(tasks)} tasks, max depth
{max_actual_depth}")
224     return True
225
226
227
228
229 class LocalObservationExtractor:
230     """
231     Extract local observation for a single agent.
232     All normalisation divisors are derived from scenario-specific parameters
233     so that observations are always in approximately [-1, 1].
234     """
235
236     def __init__(self, config: RLConfig, dataset_config: DatasetConfig,
237                  scenario: str = 'warehouse'):
238         self.config = config
239         self.dataset_config = dataset_config
240         self.scenario = scenario
241
242         # Pull scenario-specific params once at construction
243         rp = dataset_config.get_robot_params(scenario)
244         tp = dataset_config.get_task_params(scenario)
245
246         self.obs_radius = rp['observation_radius']

```

```

247         self.max_speed      = rp['max_speed']
248         self.max_payload    = rp['max_payload']
249         self.battery_cap    = rp['battery_capacity']
250         self.robot_radius   = rp['radius']
251         self.max_robots     = config.MAX_ROBOTS
252
253         # Grid half-size used for position normalisation
254         if scenario == 'warehouse':
255             self.grid_half = max(dataset_config.WAREHOUSE_GRID_SIZE) / 2.0
256         elif scenario == 'search_and_rescue':
257             self.grid_half = max(dataset_config.SAR_GRID_SIZE) / 2.0
258         else:
259             self.grid_half = max(dataset_config.MARITIME_GRID_SIZE) / 2.0
260
261         self.max_task_payload = tp['payload_weight_range'][1]
262         self.max_deadline     = tp['deadline_range'][1]
263
264         def extract(self, global_state: Dict, agent_id: int) -> np.ndarray:
265             """
266             Extract local observation for agent.
267
268             Returns:
269             local_obs: (local_obs_dim,) array containing:
270             - Own state (position, velocity, battery, payload, etc.)
271             - Nearby robots within observation radius
272             - Nearby tasks within observation radius
273             - Nearby obstacles within observation radius
274             """
275             robots      = global_state['robots']
276             tasks       = global_state['tasks']
277             obstacles   = global_state.get('obstacles', [])
278
279             own_robot   = robots[agent_id]
280             own_pos      = np.array(own_robot['position'])

```

```

281
282     local_obs = []
283
284     # Own state (15 features)
285     own_state = [
286         own_robot['position'][0] / self.grid_half,
287         own_robot['position'][1] / self.grid_half,
288         own_robot['velocity'][0] / self.max_speed,
289         own_robot['velocity'][1] / self.max_speed,
290         own_robot['max_speed'] / self.max_speed,
291         own_robot['max_payload'] / max(self.max_payload, 1.0),
292         own_robot['current_battery'] / max(self.battery_cap, 1.0),
293         own_robot['current_payload'] / max(self.max_payload, 1.0),
294         own_robot['task_completion_rate'],
295         1.0 if own_robot['operational_status'] == ROBOT_STATUS_ACTIVE
296     else 0.0,
297         1.0 if own_robot['current_task'] is not None else 0.0,
298         own_robot.get('radius', self.robot_radius) /
299         max(self.robot_radius * 2, 1.0),
300         agent_id / max(self.max_robots - 1, 1),    # FIX 3: normalised
301         by MAX_ROBOTS
302         0.0, 0.0 # Padding
303     ]
304     local_obs.extend(own_state)
305
306     # Nearby robots (max 5, each with 6 features)
307     nearby_robots = []
308     for i, robot in enumerate(robots):
309         if i != agent_id:
310             pos = np.array(robot['position'])
311             dist = np.linalg.norm(pos - own_pos)
312             if dist <= self.obs_radius:
313                 nearby_robots.append((dist, i, robot))
314     nearby_robots.sort(key=lambda x: x[0])

```

```

312
313     for i in range(5):
314         if i < len(nearby_robots):
315             dist, _, robot = nearby_robots[i]
316             rel_pos = np.array(robot['position']) - own_pos
317             features = [
318                 rel_pos[0] / self.obs_radius,
319                 rel_pos[1] / self.obs_radius,
320                 robot['velocity'][0] / self.max_speed,
321                 robot['velocity'][1] / self.max_speed,
322                 1.0 if robot['current_task'] is not None else 0.0,
323                 dist / self.obs_radius
324             ]
325         else:
326             features = [0.0] * 6
327         local_obs.extend(features)
328
329     # Nearby tasks (max 8, each with 8 features)
330     nearby_tasks = []
331     for task in tasks:
332         if task['status'] != TASK_STATUS_COMPLETE:
333             task_pos = np.array(task['location'])
334             dist = np.linalg.norm(task_pos - own_pos)
335             if dist <= self.obs_radius:
336                 nearby_tasks.append((dist, task))
337     nearby_tasks.sort(key=lambda x: x[0])
338
339     priority_map = {'high': 1.0, 'medium': 0.5, 'low': 0.25}
340
341     for i in range(8):
342         if i < len(nearby_tasks):
343             dist, task = nearby_tasks[i]
344             rel_pos = np.array(task['location']) - own_pos
345             features = [

```

```

346         rel_pos[0] / self.obs_radius,
347         rel_pos[1] / self.obs_radius,
348         priority_map.get(task['priority'], 0.5),
349         task['payload_weight'] / max(self.max_task_payload,
1.0),
350         task['deadline'] / max(self.max_deadline, 1.0),
351         1.0 if task['status'] == TASK_STATUS_IN_PROGRESS else
0.0,
352         len(task.get('dependencies', [])) / 5.0,
353         dist / self.obs_radius
354     ]
355     else:
356         features = [0.0] * 8
357         local_obs.extend(features)
358
359     # Nearby obstacles (max 10, each with 4 features)
360     nearby_obstacles = []
361     for obs in obstacles:
362         obs_pos = np.array(obs[:2])
363         dist = np.linalg.norm(obs_pos - own_pos)
364         if dist <= self.obs_radius:
365             nearby_obstacles.append((dist, obs))
366     nearby_obstacles.sort(key=lambda x: x[0])
367
368     for i in range(10):
369         if i < len(nearby_obstacles):
370             dist, obs = nearby_obstacles[i]
371             rel_pos = np.array(obs[:2]) - own_pos
372             features = [
373                 rel_pos[0] / self.obs_radius,
374                 rel_pos[1] / self.obs_radius,
375                 obs[2] / max(self.robot_radius * 10, 1.0),
376                 dist / self.obs_radius
377             ]

```

```

378         else:
379             features = [0.0] * 4
380             local_obs.extend(features)
381
382         return np.array(local_obs, dtype=np.float32)
383
384     @property
385     def local_obs_dim(self) -> int:
386         return 15 + 5 * 6 + 8 * 8 + 10 * 4    # 15 + 30 + 64 + 40 = 149
387
388
389
390 class GlobalStateExtractor:
391     """
392     Extract global state for centralized critic.
393     All normalisation divisors are scenario-specific to keep values in
394     [-1,1].
395     """
396
397     def __init__(self, config: RLConfig, max_robots: int = 20,
398                 max_tasks: int = 20, scenario: str = 'warehouse',
399                 dataset_config: DatasetConfig = None,
400                 max_steps: int = 1200):
401
402         self.config      = config
403         self.max_robots  = max_robots
404         self.max_tasks   = max_tasks
405         self.max_steps   = max_steps
406
407         dc = dataset_config or globals().get('dataset_config')
408
409         rp = dc.get_robot_params(scenario) if dc else {}
410         tp = dc.get_task_params(scenario)  if dc else {}
411
412         self.max_speed    = rp.get('max_speed', 10.0)
413         self.battery_cap  = rp.get('battery_capacity', 100.0)

```

```

411         self.max_payload = rp.get('max_payload', 200.0)
412
413         if scenario == 'warehouse':
414             self.grid_half = max(dc.WAREHOUSE_GRID_SIZE) / 2.0 if dc else
25.0
415         elif scenario == 'search_and_rescue':
416             self.grid_half = max(dc.SAR_GRID_SIZE) / 2.0 if dc else 50.0
417         else:
418             self.grid_half = max(dc.MARITIME_GRID_SIZE) / 2.0 if dc else
250.0
419
420         self.max_task_payload = tp.get('payload_weight_range', (0, 50))[1]
421         self.max_deadline = tp.get('deadline_range', (0, 100))[1]
422
423     def extract(self, global_state: Dict) -> np.ndarray:
424         robots = global_state['robots']
425         tasks = global_state['tasks']
426
427         global_obs = []
428
429         # All robot states (max_robots × 8 features)
430         for i in range(self.max_robots):
431             if i < len(robots):
432                 r = robots[i]
433                 features = [
434                     r['position'][0] / self.grid_half,
435                     r['position'][1] / self.grid_half,
436                     r['velocity'][0] / self.max_speed,
437                     r['velocity'][1] / self.max_speed,
438                     r['current_battery'] / max(self.battery_cap, 1.0),
439                     r['current_payload'] / max(self.max_payload, 1.0),
440                     1.0 if r['current_task'] is not None else 0.0,
441                     1.0 # Active indicator
442                 ]

```

```

443         else:
444             features = [0.0] * 8
445             global_obs.extend(features)
446
447         # All task states (max_tasks × 6 features)
448         for i in range(self.max_tasks):
449             if i < len(tasks):
450                 t = tasks[i]
451                 priority_val = {'high': 1.0, 'medium': 0.5, 'low':
0.25}.get(t['priority'], 0.5)
452                 status_val = {
453                     TASK_STATUS_AVAILABLE: 0.0,
454                     TASK_STATUS_IN_PROGRESS: 0.5,
455                     TASK_STATUS_COMPLETE: 1.0,
456                     TASK_STATUS_UNAVAILABLE: 0.0,
457                 }.get(t['status'], 0.0)
458                 features = [
459                     t['location'][0] / self.grid_half,
460                     t['location'][1] / self.grid_half,
461                     priority_val,
462                     t['payload_weight'] / max(self.max_task_payload,
1.0),
463                     t['deadline'] / max(self.max_deadline, 1.0),
464                     status_val
465                 ]
466             else:
467                 features = [0.0] * 6
468                 global_obs.extend(features)
469
470         # Global statistics (10 features)
471         completed = sum(1 for t in tasks if t['status'] ==
TASK_STATUS_COMPLETE)
472         assigned = sum(1 for t in tasks if t['status'] ==
TASK_STATUS_IN_PROGRESS)

```



```

473         avg_batt = (np.mean([r['current_battery'] for r in robots])
474                     / max(self.battery_cap, 1.0)) if robots else 0.0
475
476     global_stats = [
477         len(robots) / max(self.max_robots, 1),
478         len(tasks) / max(self.max_tasks, 1),
479         completed / max(len(tasks), 1),
480         assigned / max(len(tasks), 1),
481         avg_batt,
482         global_state.get('current_step', 0) / max(self.max_steps, 1),
483         0.0, 0.0, 0.0, 0.0 # Padding
484     ]
485     global_obs.extend(global_stats)
486
487     return np.array(global_obs, dtype=np.float32)
488
489     @property
490     def global_obs_dim(self) -> int:
491         return self.max_robots * 8 + self.max_tasks * 6 + 10
492
493
494     # AGENT COMMUNICATION (Multi-head attention for inter-agent coordination)

```

Listing B.5: Observation extractors that build the global, flat and local observation tensors from raw environment state.

```

1  """
2  environment/reward_shaping.py
3  =====
4  Scenario-aware reward shaper, TA-RWARE delivery reward structure.
5
6  Completion reward dominates step penalty so agents move purposefully
   toward
7  tasks rather than freezing to avoid small collision costs. Scales are set

```

```

8 per scenario: completion_reward >> step_penalty (after TA-RWARE "+1 per
9 delivery, -0.001 per step").
10
11 Deadlock detection re-assigns robots that have not moved for
    STUCK_THRESHOLD
12 steps, and a cooperative bonus fires when multiple tasks complete within a
13 short time window.
14 """
15 import numpy as np
16 from typing import Dict, Any, Optional, List, Tuple
17 from config.base_config import (
18     SCENARIO_CONFIGS, ScenarioConfig, TASK_STATUS_COMPLETE,
19     TASK_STATUS_UNAVAILABLE, TASK_STATUS_AVAILABLE, ROBOT_STATUS_FAILED,
20 )
21
22
23 class RewardShapingWrapper:
24     """
25     Per-scenario reward shaper applied to raw env rewards.
26     Wraps the environment's step output before returning to the RL agent.
27     """
28
29     def __init__(self, scenario: str):
30         if scenario not in SCENARIO_CONFIGS:
31             raise ValueError(f'Unknown scenario: {scenario}')
32         self.cfg: ScenarioConfig = SCENARIO_CONFIGS[scenario]
33         self.scenario = scenario
34         self._prev_completed = 0
35         self._stuck_steps: Dict[int, int] = {}    # robot_id steps with
no progress
36         self._stuck_best_d: Dict[int, float] = {} # robot_id best
distance so far
37         self._STUCK_THRESH = 30
38         # Dense reward: track previous "swarm-to-task" distance for

```

```

39         progress bonus
40         self._prev_sum_min_dist: Optional[float] = None
41         # Exploration: count-based intrinsic bonus on swarm centroid cell.
42         # reward_wrapper.py exploration is disabled (build_env returns
43         raw env);
44         # these bonuses fill that gap so all scenarios have exploration
45         incentive.
46         self._grid_half = self.cfg.grid_half
47         self._expl_cell_sz = max(self._grid_half * 2 / 20, 1.0)
48         self._visit_counts: Dict[Tuple[int, int], int] = {}
49         _EXPL_BONUS = {'warehouse': 0.5, 'search_and_rescue': 1.0,
50         'maritime_security': 1.5}
51         _INTR_BETA = {'warehouse': 0.02, 'search_and_rescue': 0.05,
52         'maritime_security': 0.08}
53         self._expl_bonus = _EXPL_BONUS.get(scenario, 0.5)
54         self._intrinsic_beta = _INTR_BETA.get(scenario, 0.02)
55
56     def reset(self):
57         self._prev_completed = 0
58         self._stuck_steps.clear()
59         self._stuck_best_d.clear()
60         self._prev_sum_min_dist = None
61         self._visit_counts.clear()
62
63     def shape(self, raw_reward: float, info: Dict[str, Any],
64               robots: List[Dict], tasks: List[Dict]) -> float:
65         """
66         Replace raw env reward with TA-RWARE-style delivery reward PLUS a dense
67         distance-based progress signal that bootstraps learning for SAR/Maritime.
68
69         Signal components:
70         1. Completion reward (+20/+50/+80) - strong sparse signal
71         2. Step penalty (-0.001) - encourages efficiency
72         3. Collision penalty (-0.5/-0.05/-0.03) - reduced, explores freely

```

```

68 4. Battery-fail penalty (-0.02 per failed robot) - tiny, was -0.5
69 5. NEW: Dense distance-progress bonus - positive reward when the
70 swarm gets closer to pending tasks. Scaled to ~10% of completion
71 reward magnitude so it guides exploration without dominating.
72 """
73     cfg = self.cfg
74
75     # (1) Sparse completion signal
76     n_completed = sum(1 for t in tasks if t.get('status') ==
TASK_STATUS_COMPLETE)
77     new_completions = n_completed - self._prev_completed
78     self._prev_completed = n_completed
79
80     reward = new_completions * cfg.task_completion_reward
81     reward += cfg.step_penalty
82
83     # (2) Collision (reduced)
84     if info.get('collision', False):
85         reward += cfg.collision_penalty
86
87     # (3) Battery-fail penalty (was -0.5, now -0.02)
88     # Prior value dominated the reward signal (80 failed robot-steps
= -40).
89     # New value is 25x smaller, puts it in line with step_penalty
magnitude.
90     n_failed = 0
91     for robot in robots:
92         if robot.get('operational_status') == ROBOT_STATUS_FAILED:
93             n_failed += 1
94     reward -= 0.02 * n_failed
95
96     # (4) NEW: dense distance-progress reward
97     # Compute sum over operational robots of (distance to nearest
pending task).

```

```

98         # If this sum decreased since last step, the swarm made progress
99         reward it.
100         # Scales to ~10% of completion reward magnitude.
101         # Exclude UNAVAILABLE tasks: robots cannot be assigned to them
102         yet, so
103         # including them creates penalty spikes every time a dependency
104         unlocks
105         # (task suddenly enters the set at full map distance).
106         pending_task_locs = [t['location'] for t in tasks
107                               if t.get('status') not in
108                               (TASK_STATUS_COMPLETE,
109                               TASK_STATUS_UNAVAILABLE)
110                               and 'location' in t]
111         if pending_task_locs:
112             tpos = np.asarray(pending_task_locs, dtype=np.float32) #
113             (N_tasks, 2)
114             sum_min_d = 0.0
115             n_active = 0
116             for robot in robots:
117                 if robot.get('operational_status') == ROBOT_STATUS_FAILED:
118                     continue
119                 if 'position' not in robot:
120                     continue
121                 rpos = np.asarray(robot['position'][:2], dtype=np.float32)
122                 d = float(np.min(np.linalg.norm(tpos - rpos, axis=1)))
123                 sum_min_d += d
124                 n_active += 1
125
126             if n_active > 0:
127                 # Normalise by fleet size so the signal is per-robot, not
128                 aggregate.
129                 # Avoids reward scaling with number of operational robots.
130                 mean_min_d = sum_min_d / n_active

```

```

125         if self._prev_sum_min_dist is not None:
126             improvement = self._prev_sum_min_dist - mean_min_d
127             # Scale so a full unit of per-robot progress ≈ 10% of
completion reward
128             progress_scale = max(0.003, 0.1 *
129         cfg.task_completion_reward / (cfg.grid_half * 20))
129             # Tighter negative clip (-0.2) prevents
dependency-unlock spikes
130             # from drowning the completion signal; positive cap
unchanged.
131             reward += np.clip(improvement * progress_scale, -0.2,
132         2.0)
132             self._prev_sum_min_dist = mean_min_d
133
134         # (5) Cooperative bonus: all tasks done
135         n_total = len(tasks)
136         if n_total > 0 and n_completed == n_total:
137             reward += cfg.task_completion_reward * n_total * 0.5
138
139         # (6) Joint team progress signal
140         # Per-step bonus proportional to fraction of tasks completed by
the whole
141         # team. Provides a continuous team-level signal so the
joint-reward
142         # (total tasks finished by the whole system) is always visible to
the
143         # policy --- not just at completion events.
144         if n_total > 0:
145             reward += cfg.task_completion_reward * 0.005 * (n_completed /
146         n_total)
146
147         # (7) Idle-robot penalty
148         # Small per-idle-robot charge when AVAILABLE tasks exist but some
robots

```

```

149         # have no assignment. Prevents lazy-agent collapse where a subset
150         of
151         # robots free-ride while others do all the work.
152         n_available = sum(1 for t in tasks if t.get('status') ==
TASK_STATUS_AVAILABLE)
153         if n_available > 0:
154             n_idle = sum(
155                 1 for r in robots
156                 if r.get('operational_status') != ROBOT_STATUS_FAILED
157                 and r.get('current_task') is None
158             )
159             reward -= 0.002 * n_idle
160
161         # (8) Exploration: count-based intrinsic bonus on swarm centroid
162         cell
163         # Fills the gap left by reward_wrapper.py being disabled in
164         build_env.
165         # All scenarios now have an exploration incentive; warehouse
166         benefits
167         # most since its zero-exploration setting encouraged lazy agents.
168         active_pos = [np.asarray(r['position'][:2]) for r in robots
169             if r.get('operational_status') !=
ROBOT_STATUS_FAILED
170             and 'position' in r]
171         if active_pos:
172             centroid = np.mean(active_pos, axis=0)
173             ci = int((centroid[0] + self._grid_half) / self._expl_cell_sz)
174             cj = int((centroid[1] + self._grid_half) / self._expl_cell_sz)
175             cell: Tuple[int, int] = (ci, cj)
176             cnt = self._visit_counts.get(cell, 0) + 1
177             self._visit_counts[cell] = cnt
178             if cnt == 1:
179                 reward += self._expl_bonus
180             if self._intrinsic_beta > 0:

```

```

177         reward += self._intrinsic_beta / np.sqrt(cnt)
178
179     return reward
180
181     def check_deadlock(self, robot_id: int, robot: Dict,
182                       tasks: List[Dict]) -> bool:
183         """
184         TA-RWARE deadlock detection.
185         Returns True if this robot is stuck and should be reassigned.
186         """
187         assigned_task_id = robot.get('current_task')
188         if assigned_task_id is None:
189             self._stuck_steps[robot_id] = 0
190             return False
191
192         # Find distance to assigned task
193         task = next((t for t in tasks if t.get('id') ==
194 assigned_task_id), None)
195         if task is None or task.get('status') == TASK_STATUS_COMPLETE:
196             return False
197
198         pos = np.asarray(robot['position'])
199         tpos = np.asarray(task['location'])
200         dist = float(np.linalg.norm(pos - tpos))
201
202         prev_best = self._stuck_best_d.get(robot_id, dist)
203         if dist < prev_best - 0.08 * max(prev_best, 1.0):
204             # Progress made
205             self._stuck_best_d[robot_id] = dist
206             self._stuck_steps[robot_id] = 0
207             return False
208         else:
209             # No progress
210             self._stuck_steps[robot_id] = self._stuck_steps.get(robot_id,

```



```

    0) + 1
210         self._stuck_best_d[robot_id] = min(prev_best, dist)
211
212         return self._stuck_steps.get(robot_id, 0) >= self._STUCK_THRESH

```

Listing B.6: Reward shaping: TA-RWARE delivery reward, collision penalty, deadlock penalty and coalition bonus.

```

1  """
2  environment/reward_wrapper.py
3  =====
4  Self-contained reward shaping wrapper for ImprovedTaskAllocationEnv.
5  """
6  from __future__ import annotations
7  import numpy as np
8  from typing import Dict, Tuple, Any, Optional
9  from config.base_config import TASK_STATUS_AVAILABLE
10 from environment.env_config import RLConfig as _RLCfg
11
12 SCENARIO_CFG: Dict[str, Dict] = {
13     'warehouse': {
14         'collision_scale': 1.0,
15         'completion_scale': 1.0,
16         'shaping_weight': 0.3,
17         'exploration_bonus': 0.0,
18         'intrinsic_beta': 0.0,
19         'curriculum_thresh': 0.08,
20     },
21     'search_and_rescue': {
22         'collision_scale': 0.2,
23         'completion_scale': 3.0,
24         'shaping_weight': 0.5,
25         'exploration_bonus': 2.0,
26         'intrinsic_beta': 0.05,

```

```

27         'curriculum_thresh': 0.1,
28     },
29     'maritime_security': {
30         'collision_scale': 0.15,
31         'completion_scale': 5.0,
32         'shaping_weight': 0.6,
33         'exploration_bonus': 3.0,
34         'intrinsic_beta': 0.1,
35         'curriculum_thresh': 0.05,
36     },
37 }
38
39 # Fallback grid halves matching updated SCENARIO_CONFIGS (50/150/500
    grids).
40 _GRID_HALF = {'warehouse': 25.0, 'search_and_rescue': 50.0,
41               'maritime_security': 250.0}
42
43 class RewardShapingWrapper:
44     """
45     Wraps ImprovedTaskAllocationEnv with three-layer reward shaping:
46     1. Per-scenario collision/completion rescaling
47     2. Potential-based shaping
48     3. Count-based intrinsic exploration bonus
49     """
50
51     def __init__(self, env: Any, scenario: str, gamma: float = 0.99):
52         self.env = env
53         self.scenario = scenario
54         if scenario not in SCENARIO_CFG:
55             raise ValueError(f'Unknown scenario: {scenario}')
56         self.cfg = SCENARIO_CFG[scenario]
57         self.gamma = gamma
58         self._grid_half = float(getattr(

```

```

59         getattr(env, 'local_obs_extractor', None), 'grid_half',
60         _GRID_HALF.get(scenario, 10.0)))
61     self._cell_sz = max(self._grid_half * 2 / 20, 1.0)
62     self._visits: Dict[Tuple[int,int], int] = {}
63     self._prev_phi = 0.0
64     self._sync()
65
66     def _sync(self):
67         for a in ('num_robots', 'num_tasks', 'robots', 'tasks',
68                 'observation_space', 'action_space',
69                 'local_obs_extractor', 'global_obs_extractor',
70                 'environment', 'dynamic_obstacles',
71                 'completed_tasks', 'task_assignments',
72                 'max_robots', 'max_tasks', 'grid_half'):
73             try: setattr(self, a, getattr(self.env, a))
74             except AttributeError: pass
75
76     def __getattr__(self, name):
77         return getattr(self.env, name)
78
79     def reset(self, seed=None, **kw):
80         r = self.env.reset(seed=seed, **kw) if seed is not None else
81         self.env.reset(**kw)
82         self._visits.clear(); self._sync()
83         self._prev_phi = self._phi(); self._record_visit()
84         return r
85
86     def step(self, action):
87         nxt, reward, term, trunc, info = self.env.step(action)
88         self._sync()
89         _cfg = getattr(self.env, 'config', None)
90         P_COL = getattr(_cfg, 'PENALTY_COLLISION',
91             _RLCfig.PENALTY_COLLISION)
92         R_COMP = getattr(_cfg, 'REWARD_TASK_COMPLETION',

```

```

_RLCfg.REWARD_TASK_COMPLETION)
91     if info.get('collision', False): reward +=
(self.cfg['collision_scale'] - 1.0) * P_COL
92     if info.get('task_completed', False): reward +=
(self.cfg['completion_scale'] - 1.0) * R_COMP
93     curr = self._phi()
94     reward += self.cfg['shaping_weight'] * (self.gamma * curr -
self._prev_phi)
95     self._prev_phi = curr
96     if self.cfg['intrinsic_beta'] > 0 or
self.cfg['exploration_bonus'] > 0:
97         cell = self._cell()
98         count = self._visits.get(cell, 0) + 1
99         self._visits[cell] = count
100         if count == 1: reward += self.cfg['exploration_bonus']
101         reward += self.cfg['intrinsic_beta'] / np.sqrt(count)
102         # DCMRTA improvement C: track abandoned_agent count in info dict
103         # Agents that gave up waiting > COALITION_MAX_WAITING_TIME are
counted
104         # This feeds into efficiency penalty: abandoned = wasted
coalition effort
105         n_abandoned = 0
106         tasks = getattr(self.env, 'tasks', [])
107         for t in tasks:
108             n_abandoned += len(t.get('abandoned_agent', []))
109         info['abandoned_agent_count'] = n_abandoned
110
111         # DCMRTA improvement B: coalition patience reward shaping
112         # Penalise episodes where many agents abandon tasks (>
max_waiting_time)
113         if n_abandoned > 0:
114             reward -= 0.1 * n_abandoned # small penalty per abandonment
115
116         # TA-RWARE improvement J: track idle time for warehouse

```

```

117         picker/AGV agents
118         if self.scenario == 'warehouse':
119             robots = getattr(self.env, 'robots', [])
120             n_r     = len(robots)
121             if n_r > 0:
122                 # Agents at their current position without active task =
123                 idle
124                 assigned_ids = set(getattr(self.env, 'task_assignments',
125 {})).values())
126                 n_idle = sum(1 for r in robots
127                             if r.get('id') not in assigned_ids) if
128 robots else 0
129                 info.setdefault('agvs_idle_time', n_idle)
130                 info.setdefault('pickers_idle_time', 0) # no pickers in
131 AQUA model
132         return nxt, reward, term, trunc, info
133
134     def _phi(self) -> float:
135         assigned = getattr(self.env, 'task_assignments', {})
136         completed = getattr(self.env, 'completed_tasks', set())
137         tasks     = getattr(self.env, 'tasks', [])
138         robots    = getattr(self.env, 'robots', [])
139         pending   = [t for t in tasks
140                     if t.get('id') not in assigned and
141                        t.get('id') not in completed and
142                        t.get('status', TASK_STATUS_AVAILABLE) ==
143 TASK_STATUS_AVAILABLE]
144         if not pending: return 1.0
145         if not robots:  return 0.0
146         total = sum(
147             min(np.linalg.norm(
148                 np.asarray(r.get('position', [0., 0.])) -
149                 np.asarray(t.get('location', [0., 0.]))
150             ) for t in pending) / max(self._grid_half, 1.0)

```

```

144         for r in robots)
145     return -total / max(len(robots),1)
146
147     def _cell(self) -> Tuple[int,int]:
148         robots = getattr(self.env,'robots',[])
149         if not robots: return (0,0)
150         pos = np.mean([np.asarray(r.get('position',[0.,0.])) for r in
151 robots], axis=0)
152         return (int((pos[0]+self._grid_half)/self._cell_sz),
153                 int((pos[1]+self._grid_half)/self._cell_sz))
154
155     def _record_visit(self):
156         c = self._cell(); self._visits[c] = self._visits.get(c,0) + 1
157
158     def get_state_dim(self):          return self.env.get_state_dim()
159     def get_action_dim(self):         return self.env.get_action_dim()
160     def get_local_observations(self): return
161         self.env.get_local_observations()
162     def get_global_observation(self): return
163         self.env.get_global_observation()
164
165     def get_action_mask(self):        return self.env.get_action_mask()
166     def get_global_state(self):       return self.env.get_global_state()
167
168
169
170
171
172
173

```

```

144         for r in robots)
145     return -total / max(len(robots),1)
146
147     def _cell(self) -> Tuple[int,int]:
148         robots = getattr(self.env,'robots',[])
149         if not robots: return (0,0)
150         pos = np.mean([np.asarray(r.get('position',[0.,0.])) for r in
151 robots], axis=0)
152         return (int((pos[0]+self._grid_half)/self._cell_sz),
153                 int((pos[1]+self._grid_half)/self._cell_sz))
154
155     def _record_visit(self):
156         c = self._cell(); self._visits[c] = self._visits.get(c,0) + 1
157
158     def get_state_dim(self):          return self.env.get_state_dim()
159     def get_action_dim(self):         return self.env.get_action_dim()
160     def get_local_observations(self): return
161         self.env.get_local_observations()
162     def get_global_observation(self): return
163         self.env.get_global_observation()
164
165     def get_action_mask(self):        return self.env.get_action_mask()
166     def get_global_state(self):       return self.env.get_global_state()
167
168
169
170
171
172
173

```

```

165 def make_curriculum(scenario: str):
166     try:
167         from environment.task_env import CurriculumManager
168         from config.base_config import SCENARIO_CONFIGS
169         cm =
170 CurriculumManager(threshold=SCENARIO_CONFIGS[scenario].curriculum_threshold)
171         return cm
172     except ImportError:
173         return None

```

```

174
175 def build_env(scenario: str, seed: int = 42):
176     """
177     Single entry point - returns ImprovedTaskAllocationEnv with PPG +
        CurriculumManager.
178
179     Difficulty managers wired in:
180     - PerformancePacedGenerator (PPG): continuous Goldilocks-Zone adjustment
        (40-80%
181     success). Overrides CurriculumManager difficulty ranges each episode.
182     - CurriculumManager: staged 5-level progression (L0→L4). Records
        outcomes and
183     advances levels; its difficulty ranges are overridden by PPG but level
        tracking
184     remains active for logging.
185     Both are updated each episode via env.record_episode_outcome(), which is
        called
186     by train_rl_agent after every episode.
187
188     Reward shaping note: the training loop applies TA-RWARE shaping via
189     environment/reward_shaping.py::RewardShapingWrapper.shape(), which
        overwrites raw
190     env rewards. Wrapping the env in reward_wrapper.RewardShapingWrapper
        here would
191     double-apply shaping, so the base env is returned unwrapped.
192     """
193     from environment.task_env import ImprovedTaskAllocationEnv,
        PerformancePacedGenerator
194     from environment.env_config import rl_config, dataset_config
195     ppg = PerformancePacedGenerator(scenario)
196     cm = make_curriculum(scenario)
197     base = ImprovedTaskAllocationEnv(
198         scenario=scenario, config=rl_config,
199         dataset_config_param=dataset_config,

```

```

200         curriculum_manager=cm,
201         performance_generator=ppg,
202     )
203     return base

```

Listing B.7: Reward wrapper that applies the shaping function over the base environment reward signal.

```

1  """
2  task_env.py
3  =====
4  ImprovedTaskAllocationEnv and supporting classes.
5
6  Core multi-robot task allocation Gymnasium environment used by all three
7  training scenarios (warehouse, search_and_rescue, maritime_security).
8
9  Arrival tolerance is scenario-scaled: max(robot_radius * 1.5, grid_half *
    0.02).
10 This prevents maritime vessels (radius 3 m) from never reaching a 0.5 m
    target
11 and warehouse AMRs (radius 0.4 m) from getting an oversized approach
    window.
12 """
13 import os
14 import sys
15 import gc
16 import pickle
17 import yaml
18 import time
19 import math
20 import heapq
21 import numpy as np
22 import pandas as pd
23 import torch

```



```

24 import torch.nn as nn
25 import torch.nn.functional as F
26 import torch.optim as optim
27 from torch.optim.lr_scheduler import CosineAnnealingLR, LinearLR,
    OneCycleLR
28 from torch.distributions import Categorical, Normal
29 try:
30     from torch.amp import autocast, GradScaler          # PyTorch >= 2.0
31 except ImportError:
32     from torch.cuda.amp import autocast, GradScaler     # PyTorch < 2.0
33 import matplotlib
34 matplotlib.use('Agg') # Non-interactive backend - required for headless
    servers
35 import matplotlib.pyplot as plt
36 from pathlib import Path
37 from typing import List, Dict, Tuple, Optional, Any, Union, Callable,
    NamedTuple
38 from tqdm.auto import tqdm
39 import json
40 from datetime import datetime
41 from collections import deque, defaultdict
42 from abc import ABC, abstractmethod
43 from dataclasses import dataclass, field
44 from scipy import stats
45 from scipy.optimize import linear_sum_assignment
46 import warnings
47 import copy
48 import logging
49 logger = logging.getLogger(__name__)
50 import traceback
51 import threading
52 from contextlib import contextmanager
53 from functools import lru_cache
54 import random

```

```

55 import seaborn as sns
56
57 warnings.filterwarnings('ignore')
58
59 import gymnasium as gym
60 from gymnasium import spaces
61
62 # Import config singletons from env_config (defined in same package)
63 from environment.env_config import (
64     DatasetConfig, RLConfig,
65     dataset_config, rl_config
66 )
67 from environment.observation_extractors import LocalObservationExtractor,
68     GlobalStateExtractor
69 from config.base_config import (
70     SCENARIO_CONFIGS,
71     SCENARIO_TASK_TYPES,
72     TASK_STATUS_UNAVAILABLE,
73     TASK_STATUS_AVAILABLE,
74     TASK_STATUS_IN_PROGRESS,
75     TASK_STATUS_COMPLETE,
76     DEPENDENT_TASK_REWARD_MULTIPLIER,
77     COALITION_MAX_WAITING_TIME,
78     ROBOT_STATUS_ACTIVE,
79     ROBOT_STATUS_INACTIVE,
80     ROBOT_STATUS_FAILED,
81 )
82
83 from utils.action_masking import StuckCounter
84
85 # Simulation parameters (inline - not in env_config)
86 class _SimConfig:
87     TASK_TRAVEL_REQUIRED = True
88     DYNAMIC_OBSTACLE_PROBABILITY = 0.3
89     NUM_DYNAMIC_OBSTACLES = 3

```

```

88     DYNAMIC_OBSTACLE_SPEED_RANGE = (0.1, 0.5)
89
90     sim_config = _SimConfig()
91
92
93     class DynamicObstacle:
94         """Dynamic obstacle with prediction capability"""
95
96         def __init__(self, position: np.ndarray, velocity: np.ndarray,
97             radius: float):
98             self.position = position.copy()
99             self.velocity = velocity.copy()
100             self.radius = radius
101             self.trajectory = [position.copy()]
102
103         def update(self, dt: float, bounds: Tuple[float, float, float, float]
104             = None):
105             """Update position"""
106             self.position += self.velocity * dt
107
108             if bounds is not None:
109                 x_min, x_max, y_min, y_max = bounds
110                 if self.position[0] < x_min or self.position[0] > x_max:
111                     self.velocity[0] *= -1
112                 if self.position[1] < y_min or self.position[1] > y_max:
113                     self.velocity[1] *= -1
114                 self.position[0] = np.clip(self.position[0], x_min, x_max)
115                 self.position[1] = np.clip(self.position[1], y_min, y_max)
116
117             self.trajectory.append(self.position.copy())
118
119         def predict_positions(self, horizon: int, dt: float) -> np.ndarray:
120             """Predict future positions"""
121             positions = np.zeros((horizon, 2))

```

```

120         pos = self.position.copy()
121         vel = self.velocity.copy()
122
123         for t in range(horizon):
124             positions[t] = pos
125             pos = pos + vel * dt
126
127         return positions
128
129
130
131 class EnvironmentGenerator:
132     """Generate environments for different scenarios"""
133
134     def __init__(self, config: DatasetConfig):
135         self.config = config
136
137     def generate_environment(self, scenario: str) -> Dict:
138         """Generate environment based on scenario"""
139         if scenario == 'warehouse':
140             return self._generate_warehouse()
141         elif scenario == 'search_and_rescue':
142             return self._generate_sar()
143         elif scenario == 'maritime_security':
144             return self._generate_maritime()
145         else:
146             raise ValueError(f"Unknown scenario: {scenario}")
147
148     def _generate_warehouse(self) -> Dict:
149         grid_size = self.config.WAREHOUSE_GRID_SIZE
150         num_shelves = self.config.WAREHOUSE_NUM_SHELVES
151         gx, gy = grid_size[0] / 2, grid_size[1] / 2
152
153         obstacles = []

```

```

154         # Structured aisle layout: shelves in rows with 3m corridors
        between.

155         aisle_width = 3.0
156         shelf_len    = 2.0
157         shelf_r      = 0.8
158         row_spacing = 4.0 + aisle_width
159
160         usable_y  = gy * 1.8
161         n_rows    = max(1, int(usable_y / row_spacing))
162         usable_x  = gx * 1.8
163         n_per_row = max(1, int(usable_x / (shelf_len + 1.0)))
164
165         shelf_count = 0
166         for row in range(n_rows):
167             if shelf_count >= num_shelves:
168                 break
169             y_center = -usable_y / 2 + row * row_spacing + row_spacing / 2
170             for col in range(n_per_row):
171                 if shelf_count >= num_shelves:
172                     break
173                 x_center = -usable_x / 2 + col * (shelf_len + 1.0) +
shelf_len / 2
174                 jx = np.random.uniform(-0.3, 0.3)
175                 jy = np.random.uniform(-0.3, 0.3)
176                 x = float(np.clip(x_center + jx, -gx + 1.5, gx - 1.5))
177                 y = float(np.clip(y_center + jy, -gy + 1.5, gy - 1.5))
178                 obstacles.append([x, y, shelf_r])
179                 shelf_count += 1
180
181         return {
182             'type': 'warehouse',
183             'grid_size': grid_size,
184             'obstacles': np.array(obstacles) if obstacles else
np.array([]).reshape(0, 3),

```

```

185         'dynamic_obstacles': []
186     }
187
188     def _generate_sar(self) -> Dict:
189         grid_size = self.config.SAR_GRID_SIZE
190         num_obstacles = self.config.SAR_NUM_OBSTACLES
191         gx, gy = grid_size[0] / 2, grid_size[1] / 2
192
193         obstacles = []
194         # Rubble-cluster pattern: debris radiates from collapsed
structure centres.
195         n_clusters = max(3, num_obstacles // 8)
196         per_cluster = num_obstacles // n_clusters
197         extra = num_obstacles - n_clusters * per_cluster
198
199         for ci in range(n_clusters):
200             cx = np.random.uniform(-gx * 0.8, gx * 0.8)
201             cy = np.random.uniform(-gy * 0.8, gy * 0.8)
202             spread = np.random.uniform(4.0, 12.0)
203             count = per_cluster + (1 if ci < extra else 0)
204             for _ in range(count):
205                 angle = np.random.uniform(0, 2 * np.pi)
206                 dist = np.random.exponential(spread / 3.0)
207                 x = float(np.clip(cx + dist * np.cos(angle), -gx + 1.0,
gx - 1.0))
208                 y = float(np.clip(cy + dist * np.sin(angle), -gy + 1.0,
gy - 1.0))
209                 r = np.random.uniform(1.0, 4.0)
210                 obstacles.append([x, y, r])
211
212         return {
213             'type': 'search_and_rescue',
214             'grid_size': grid_size,
215             'obstacles': np.array(obstacles) if obstacles else

```

```

216         np.array([]).reshape(0, 3),
217         'dynamic_obstacles': []
218     }
219
220     def _generate_maritime(self) -> Dict:
221         """Generate maritime security environment.
222
223         Uses all MARITIME_* config fields:
224         - MARITIME_NUM_BUOYS: navigation markers
225         - MARITIME_NUM_VESSELS: large other-ship obstacles (radius ~3 m)
226         - MARITIME_NUM_RESTRICTED_AREAS: exclusion-zone circles (radius 8-20 m)
227         - MARITIME_NUM_PATROL_ZONES: patrol-area centres stored as metadata
228         - MARITIME_CURRENT/WAVE/WIND_*: environmental conditions stored as
229           metadata
230
231         (available for downstream physics extension; not currently applied to
232         robot dynamics, which are governed by per-scenario battery params)
233
234         """
235
236         grid_size = self.config.MARITIME_GRID_SIZE
237         gx, gy = grid_size[0] / 2, grid_size[1] / 2
238
239         obstacles = []
240
241         # Fixed buoys (small navigation markers, radius 1 m)
242         for _ in range(self.config.MARITIME_NUM_BUOYS):
243             obstacles.append([
244                 np.random.uniform(-gx, gx),
245                 np.random.uniform(-gy, gy),
246                 1.0
247             ])
248
249         # Other vessels (large moving ships, radius ~3 m)
250         vessel_r = 3.0
251         for _ in range(self.config.MARITIME_NUM_VESSELS):
252             obstacles.append([

```

```

248         np.random.uniform(-gx + vessel_r, gx - vessel_r),
249         np.random.uniform(-gy + vessel_r, gy - vessel_r),
250         vessel_r
251     ])
252
253     # Restricted areas (exclusion zones, radius 20-60 m for 500m
grid)
254     for _ in range(self.config.MARITIME_NUM_RESTRICTED_AREAS):
255         r = np.random.uniform(20.0, 60.0)
256         obstacles.append([
257             np.random.uniform(-gx + r, gx - r),
258             np.random.uniform(-gy + r, gy - r),
259             r
260         ])
261
262     # Patrol zone centres (metadata - not obstacles)
263     patrol_zones = np.array([
264         [np.random.uniform(-gx, gx), np.random.uniform(-gy, gy)]
265         for _ in range(self.config.MARITIME_NUM_PATROL_ZONES)
266     ], dtype=np.float32)
267
268     # Environmental conditions (metadata for future physics
extension)
269     env_conditions = {
270         'current_strength':
271         float(np.random.uniform(*self.config.MARITIME_CURRENT_STRENGTH_RANGE)),
272         'wave_height':
273         float(np.random.uniform(*self.config.MARITIME_WAVE_HEIGHT_RANGE)),
274         'wind_speed':
275         float(np.random.uniform(*self.config.MARITIME_WIND_SPEED_RANGE)),
276     }
277
278     return {
279         'type': 'maritime_security',

```



```

277         'grid_size': grid_size,
278         'obstacles': np.array(obstacles) if obstacles else
np.array([]).reshape(0, 3),
279         'dynamic_obstacles': [],
280         'patrol_zones':      patrol_zones,
281         'env_conditions':    env_conditions,
282     }
283
284 class PerformancePacedGenerator:
285     """Self-paced curriculum: adjusts difficulty based on agent success
rate.
286
287     Keeps the agent in the Goldilocks Zone (40-80% success).
288     Difficulty is a fractional value in [0, 1] that controls robot/task
counts
289     and whether dynamic obstacles and task dependencies are enabled.
290     Hysteresis prevents rapid oscillation: at least _HYSTERESIS episodes
must
291     pass between difficulty changes.
292     """
293
294     _WINDOW      = 50
295     _HYSTERESIS = 30
296
297     def __init__(self, scenario: str):
298         self.scenario = scenario
299         sc_cfg = SCENARIO_CONFIGS[scenario]
300         self._max_robots = sc_cfg.num_robots_max
301         self._max_tasks  = sc_cfg.max_tasks
302         self._difficulty = 0.1
303         self._history: List[float] = []
304         self._steps_at_level = 0
305         # Use per-scenario threshold so SAR (0.10) and maritime (0.05) can
306         # actually advance --- the old hardcoded 0.80 made those

```

```

impossible.
307     self._high_thresh = sc_cfg.curriculum_threshold
308     self._low_thresh  = max(0.05, sc_cfg.curriculum_threshold * 0.5)
309
310 def record(self, completion_rate: float):
311     """Record episode outcome and adjust difficulty if warranted."""
312     self._history.append(completion_rate)
313     if len(self._history) > self._WINDOW:
314         self._history.pop(0)
315     self._steps_at_level += 1
316
317     if (len(self._history) >= self._WINDOW
318         and self._steps_at_level >= self._HYSTERESIS):
319         mean = float(np.mean(self._history))
320         if mean > self._high_thresh:
321             self._difficulty = min(1.0, self._difficulty + 0.05)
322             self._steps_at_level = 0
323             logger.info(
324                 f"[PPG:{self.scenario}] success={mean:.2f} >
325                 {self._high_thresh}"
326                 f" → difficulty {self._difficulty:.2f}"
327             )
328         elif mean < self._low_thresh:
329             self._difficulty = max(0.0, self._difficulty - 0.05)
330             self._steps_at_level = 0
331             logger.info(
332                 f"[PPG:{self.scenario}] success={mean:.2f} <
333                 {self._low_thresh}"
334                 f" → difficulty {self._difficulty:.2f}"
335             )
336
337 def get_params(self) -> Dict:
338     """Return episode params scaled to current difficulty fraction."""
339     d = self._difficulty

```

```

338     r_min = max(1, int(2 + d * (self._max_robots - 2) * 0.5))
339     r_max = max(r_min + 1, int(2 + d * (self._max_robots - 2)))
340     t_min = max(2, int(2 + d * (self._max_tasks - 2) * 0.5))
341     t_max = max(t_min + 1, int(2 + d * (self._max_tasks - 2)))
342     return {
343         'num_robots_range':      (r_min, r_max),
344         'num_tasks_range':      (t_min, t_max),
345         'use_dynamic_obstacles': d >= 0.4,
346         'use_dependencies':     d >= 0.6,
347         'difficulty':           d,
348     }
349
350     def state_dict(self) -> Dict:
351         return {
352             'difficulty':       self._difficulty,
353             'history':          list(self._history),
354             'steps_at_level':   self._steps_at_level,
355         }
356
357     def load_state_dict(self, state: Dict):
358         self._difficulty      = state.get('difficulty', 0.1)
359         self._history          = list(state.get('history', []))
360         self._steps_at_level  = state.get('steps_at_level', 0)
361
362
363     # CurriculumManager
364
365
366     class CurriculumManager:
367         """
368         Curriculum learning manager for progressive difficulty scaling.
369
370         Difficulty levels range from 0 (easiest) to CURRICULUM_LEVELS-1
            (hardest).

```

```

371 Advances to the next level when the rolling mean completion rate exceeds
372 CURRICULUM_THRESHOLD over the last CURRICULUM_WINDOW episodes.
373 """
374
375 def __init__(self, config: RLConfig = None, threshold: float = None):
376     self.config = config or rl_config
377     self.level = 0
378     self.max_level = self.config.CURRICULUM_LEVELS - 1
379     self.threshold = threshold if threshold is not None else
self.config.CURRICULUM_THRESHOLD
380     self.window = self.config.CURRICULUM_WINDOW
381     self.performance_history: List[float] = []
382
383     logger.info(
384         f"CurriculumManager initialized: "
385         f"levels={self.config.CURRICULUM_LEVELS}, "
386         f"threshold={self.threshold}, "
387         f"window={self.window}"
388     )
389
390
391 def get_difficulty_params(self) -> Dict:
392     """Return environment difficulty parameters for the current level.
393
394     5-level stepped curriculum --- each level has its own robot/task
395     ceiling
396     so agents face single-robot tasks before scaling to full
397     multi-robot
398     complexity. Within each level, tasks_max ramps from the base
399     toward
400     the level ceiling as rolling-mean performance rises, so the
401     curriculum
402     gets harder *before* advancing --- agents must prove they can
403     handle both

```

```

399         simple and dense task sets at each robot count.
400
401         L0: 1--2 robots, 2-- 4 tasks, no dynamics, no deps (solo
routing)
402         L1: 2--4 robots, 3-- 8 tasks, no dynamics, no deps (basic
cooperation)
403         L2: 3--8 robots, 5--15 tasks, dynamics ON, no deps
(obstacle avoidance)
404         L3: 6--15 robots, 10--30 tasks, dynamics ON, deps ON
(dependency chains)
405         L4: MAX--MAX robots, 20--MAX tasks, all features (full
scenario)
406         """
407         mr = self.config.MAX_ROBOTS # from config --- not hardcoded
408         mt = self.config.MAX_TASKS
409
410         LEVEL_PARAMS = [
411             # (r_min, r_max, t_min, t_max_ceil, dynamics, deps)
412             (1, 2, 2, 4, False, False),
413             # L0
414             (2, 4, 3, 8, False, False),
415             # L1
416             (3, max(4, mr//3), 5, 15, True, False),
417             # L2
418             (max(4, mr//3), max(8, mr*2//3), 10, 30, True, True),
419             # L3
420             (max(8, mr//2), mr, 20, mt, True, True),
421             # L4
422         ]
423
424         lv = min(self.level, len(LEVEL_PARAMS) - 1)
425         r_min, r_max, t_min, t_max_ceil, use_dyn, use_deps =
LEVEL_PARAMS[lv]
426
427         # Intra-level difficulty: tasks_max grows from t_min toward

```

```

422         # as mean completion rate improves within this level.
423         if len(self.performance_history) >= max(1, self.window // 4):
424             perf = float(np.mean(self.performance_history[-self.window:]))
425         else:
426             perf = 0.0
427         intra = min(1.0, perf / max(self.threshold, 1e-3))
428         t_max = max(t_min + 1, int(t_min + intra * (t_max_ceil - t_min)))
429
430         return {
431             'num_robots_range':      (r_min, min(r_max, mr)),
432             'num_tasks_range':      (t_min, min(t_max, mt)),
433             'use_dynamic_obstacles': use_dyn,
434             'use_dependencies':     use_deps,
435             'level':                 self.level,
436             'difficulty_fraction':   lv / max(1, len(LEVEL_PARAMS) - 1),
437         }
438
439     def record_performance(self, completion_rate: float):
440         """Record completion rate for a single episode."""
441         self.performance_history.append(completion_rate)
442         # Keep only the rolling window
443         if len(self.performance_history) > self.window:
444             self.performance_history.pop(0)
445
446     def advance(self) -> bool:
447         """
448         Check whether to advance to the next curriculum level.
449         Returns True if the level was advanced.
450         """
451         if self.level >= self.max_level:
452             return False
453
454         if len(self.performance_history) < self.window:

```

```

455         return False
456
457         mean_completion = np.mean(self.performance_history[-self.window:])
458         if mean_completion >= self.threshold:
459             self.level += 1
460             self.performance_history.clear()    # reset window for next
level
461             logger.info(
462                 f"Curriculum advanced to level
{self.level}/{self.max_level} "
463                 f"(mean completion={mean_completion:.3f} >=
{self.threshold})"
464             )
465             return True
466
467         return False
468
469         # Checkpoint support (used by RobustCheckpointManager)
470
471
472         def state_dict(self) -> Dict:
473             return {
474                 'level':            self.level,
475                 'performance_history': list(self.performance_history),
476             }
477
478         def load_state_dict(self, state: Dict):
479             self.level            = state.get('level', 0)
480             self.performance_history = list(state.get('performance_history',
481                 []))
482
483         def __repr__(self) -> str:
484             recent = (np.mean(self.performance_history[-self.window:])
485                 if self.performance_history else 0.0)

```

```

485         return (f"CurriculumManager(level={self.level}/{self.max_level}, "
486                 f"recent_mean={recent:.3f})")
487
488
489 # Convenience singleton (mirrors the pattern used by dataset_config /
         rl_config)
490 curriculum_manager = CurriculumManager()
491
492
493
494 class ImprovedTaskAllocationEnv(gym.Env):
495     """
496     Improved Multi-Robot Task Allocation Environment
497
498     Features:
499     - Realistic task execution with travel time
500     - Dynamic obstacles with prediction
501     - Inter-robot collision checking
502     - Battery and energy management
503     - Task dependencies
504     - Curriculum learning support
505     """
506
507     def __init__(self, scenario: str = 'warehouse', config: RLConfig =
        None,
508                 dataset_config_param: DatasetConfig = None,
509                 curriculum_manager: CurriculumManager = None,
510                 performance_generator: 'PerformancePacedGenerator' =
        None):
511         super().__init__()
512
513         self.config = config or rl_config
514         self.dataset_config = dataset_config_param or dataset_config
515         self.scenario = scenario

```



```

516     self.curriculum = curriculum_manager
517     self.performance_generator = performance_generator
518     self._max_steps = SCENARIO_CONFIGS[scenario].max_steps
519
520     # Scenario-specific robot parameters
521     self.robot_params = self.dataset_config.get_robot_params(scenario)
522
523     self.max_robots = self.config.MAX_ROBOTS
524     self.max_tasks = self.config.MAX_TASKS
525
526     # Observation dimensions
527     self.robot_feature_dim = 15
528     self.task_feature_dim = 12
529     self.global_feature_dim = 10
530
531     obs_dim = (self.max_robots * self.robot_feature_dim +
532                self.max_tasks * self.task_feature_dim +
533                self.global_feature_dim)
534
535     self.observation_space = spaces.Box(
536         low=-np.inf, high=np.inf, shape=(obs_dim,), dtype=np.float32
537     )
538
539     self.action_space = spaces.Discrete(self.max_robots *
self.max_tasks + 1)
540
541     # Task types per scenario
542     self.task_types = self._get_task_types()
543     self.priority_encoding = {'high': 3, 'medium': 2, 'low': 1}
544
545     # Environment generator
546     self.env_generator = EnvironmentGenerator(self.dataset_config)
547
548     # Observation extractors - scenario-aware normalisations

```

```

549     self.local_obs_extractor = LocalObservationExtractor(
550         self.config, self.dataset_config, scenario=scenario
551     )
552     self.global_obs_extractor = GlobalStateExtractor(
553         self.config, self.max_robots, self.max_tasks,
554         scenario=scenario, dataset_config=self.dataset_config,
555         max_steps=self._max_steps
556     )
557
558     # State variables
559     self.reset()
560
561     def _get_task_types(self) -> Dict[str, int]:
562         if self.scenario == 'warehouse':
563             types = self.dataset_config.TASK_TYPES_WAREHOUSE
564         elif self.scenario == 'search_and_rescue':
565             types = self.dataset_config.TASK_TYPES_SAR
566         else:
567             types = self.dataset_config.TASK_TYPES_MARITIME
568             # dataclass field returns default; ensure we have an iterable list
569             if not isinstance(types, (list, tuple)):
570                 types = list(types) if hasattr(types, '__iter__') else
571                 ['task']
572
573             return {t: i for i, t in enumerate(types)}
574
575     def reset(self, seed: int = None, options: Dict = None) ->
576     Tuple[np.ndarray, Dict]:
577         if seed is not None:
578             np.random.seed(seed)
579
580         # Get difficulty from curriculum
581         if self.curriculum:
582             diff = self.curriculum.get_difficulty_params()
583             num_robots_range = diff['num_robots_range']

```

```

581         num_tasks_range = diff['num_tasks_range']
582         self.use_dynamic_obstacles = diff['use_dynamic_obstacles']
583         self.use_dependencies = diff['use_dependencies']
584     else:
585         num_robots_range = self.robot_params['num_robots_range']
586         num_tasks_range = (5, self.max_tasks)
587         self.use_dynamic_obstacles = True
588         self.use_dependencies = True
589
590         # Performance-paced generator overrides curriculum ranges when
provided
591         if self.performance_generator is not None:
592             ppg = self.performance_generator.get_params()
593             num_robots_range = ppg['num_robots_range']
594             num_tasks_range = ppg['num_tasks_range']
595             self.use_dynamic_obstacles = ppg['use_dynamic_obstacles']
596             self.use_dependencies = ppg['use_dependencies']
597
598         # Hard cap task count to scenario maximum
599         sc_max_tasks = SCENARIO_CONFIGS[self.scenario].max_tasks
600         num_tasks_range = (num_tasks_range[0], min(num_tasks_range[1],
601             sc_max_tasks))
602
603         # Generate environment
604         self.environment =
605         self.env_generator.generate_environment(self.scenario)
606
607         # compute scenario-scaled "at target" arrival tolerance once per
# episode. The old hard-coded 0.5 m was scenario-agnostic, which
meant
608         # a 3 m maritime vessel could never be "at target" in a
numerically
609         # stable way, and warehouse 0.4 m AMRs got a bigger margin than
their

```

```

609         # own body. New rule: tolerance = max(robot_radius * 1.5,
grid_half * 0.02).
610         # warehouse (r=0.4, grid_half=7): max(0.60, 0.14) = 0.60 m
611         # SAR (r=0.65, grid_half=15): max(0.98, 0.30) = 0.98 m
612         # maritime (r=3.0, grid_half=60): max(4.50, 1.20) = 4.50 m
613         grid_size_env = self.environment['grid_size']
614         grid_half_env = max(grid_size_env) / 2.0
615         robot_r = self.robot_params['radius']
616         self._arrival_tol = max(robot_r * 1.5, grid_half_env * 0.02)
617
618         # Maritime: extract live env conditions from generated
environment
619         # env_conditions holds current_strength, wave_height, wind_speed.
620         # These are applied every step in _update_robot_positions for
maritime.
621         if self.scenario == 'maritime_security':
622             ec = self.environment.get('env_conditions', {})
623             self._maritime_current_strength =
float(ec.get('current_strength', 1.0))
624             self._maritime_wave_height      = float(ec.get('wave_height',
1.0))
625             self._maritime_wind_speed       = float(ec.get('wind_speed',
10.0))
626             # Persistent current direction - randomised once per episode
627             angle = np.random.uniform(0, 2 * np.pi)
628             self._maritime_current_dir = np.array([np.cos(angle),
np.sin(angle)],
629                                                    dtype=np.float32)
630         else:
631             self._maritime_current_strength = 0.0
632             self._maritime_wave_height      = 0.0
633             self._maritime_wind_speed       = 0.0
634             self._maritime_current_dir      = np.zeros(2,
dtype=np.float32)

```

```

635
636     # Number of agents
637     self.num_robots = np.random.randint(num_robots_range[0],
638                                         min(num_robots_range[1],
self.max_robots) + 1)
639     self.num_tasks = np.random.randint(num_tasks_range[0],
640                                         min(num_tasks_range[1],
self.max_tasks) + 1)
641
642     # Generate robots and tasks
643     self.robots = self._generate_robots()
644     self.tasks = self._generate_tasks()
645
646     # Initialize tracking
647     self.task_assignments = {}
648     self.robot_loads = {i: 0 for i in range(self.num_robots)}
649     self.completed_tasks = set()
650     # FIX: tracks tasks that already gave REWARD_TASK_COMPLETION
651     # to prevent the 'Track completed tasks' loop from re-firing
652     # every subsequent step once a task is in
task_assignments+completed_tasks.
653     # Separate from completed_tasks: prevents re-issuing the
completion
654     # reward on subsequent steps when a task stays in
task_assignments.
655     self._rewarded_task_ids: set = set()
656     self.current_step = 0
657     self.total_energy = 0.0
658     self.collisions = 0
659     self.robot_collisions = 0
660
661     # Dynamic reallocation tracking
662     self.reallocation_count = 0           # total mid-episode
reallocations

```

```

663         self.reallocation_events = []           # list of dicts describing
each event
664         self._last_realloc_step = -1           # throttle: one realloc pass
per step
665
666         # DCMRTA metrics tracking (A B C D)
667         self._robot_travel_dist = {i: 0.0 for i in
range(self.max_robots)}
668         self._robot_waiting_time = {i: 0.0 for i in
range(self.max_robots)}
669         self._robot_prev_pos = {}              # for travel_dist increment
670         self._coalition_members = {}           # task_id [robot_ids waiting]
671         self._coalition_wait_start = {}        # robot_id step it started
waiting
672         self._abandoned_agents = []           # (robot_id, task_id, reason)
673         self._task_wait_times = {}            # task_id total waiting time
674
675         # Task execution state (NEW: realistic execution)
676         self.robot_targets = {i: None for i in range(self.num_robots)}
677         self.task_progress = {i: 0.0 for i in range(self.num_tasks)}
678
679         # Dynamic obstacles
680         self.dynamic_obstacles = []
681         if self.use_dynamic_obstacles and np.random.random() <
sim_config.DYNAMIC_OBSTACLE_PROBABILITY:
682             self._generate_dynamic_obstacles()
683
684         return self._get_observation(), self._get_info()
685
686     def _generate_robots(self) -> List[Dict]:
687         robots = []
688         grid_size = self.environment['grid_size']
689         rp = self.robot_params
690         obstacles = self.environment.get('obstacles', [])

```

```

691     _MIN_SEP = 2.0 # minimum spawn clearance (metres) from
        obstacles & other robots

692
693     for i in range(self.num_robots):
694         x_lo, x_hi = -grid_size[0]/2 + 2, grid_size[0]/2 - 2
695         y_lo, y_hi = -grid_size[1]/2 + 2, grid_size[1]/2 - 2
696         for _attempt in range(30):
697             position = np.array([
698                 np.random.uniform(x_lo, x_hi),
699                 np.random.uniform(y_lo, y_hi),
700             ])
701             # Reject if inside or too close to any static obstacle
702             clear = True
703             for obs in obstacles:
704                 obs_r = float(obs[2]) if len(obs) > 2 else 1.0
705                 if np.linalg.norm(position - np.asarray(obs[:2])) <
obs_r + _MIN_SEP:
706                     clear = False
707                     break
708             if clear:
709                 # Reject if too close to an already-placed robot
710                 for placed in robots:
711                     if np.linalg.norm(position - placed['position'])
< _MIN_SEP:
712                         clear = False
713                         break
714             if clear:
715                 break
716             # After 30 tries use the last candidate rather than crashing
717
718         robot = {
719             'id': i,
720             'position': position,
721             'velocity': np.zeros(2),

```

```

722         'max_speed': rp['max_speed'] * np.random.uniform(0.85,
1.15),
723         'max_speed_loaded': rp['max_speed_loaded'] *
np.random.uniform(0.85, 1.15),
724         'max_payload': rp['max_payload'] *
np.random.uniform(0.85, 1.15),
725         'battery_capacity': rp['battery_capacity'],
726         'current_battery': np.random.uniform(
rp['battery_capacity'] * 0.75,
727         rp['battery_capacity']
728     ),
729     'battery_drain_rate': rp['battery_drain_rate'],
730     'battery_drain_loaded_factor':
rp['battery_drain_loaded_factor'],
731     'task_completion_rate': np.random.uniform(0.8, 1.0),
732     'current_payload': 0.0,
733     'operational_status': ROBOT_STATUS_ACTIVE,
734     'current_task': None,
735     'radius': rp['radius'],
736     'observation_radius': rp['observation_radius'],
737     'communication_range': rp['communication_range'],
738 }
739     robots.append(robot)
740
741
742     return robots
743
744     def _generate_tasks(self) -> List[Dict]:
745         """Generate episode tasks sampled from the scenario-specific
typed task pool.
746
747         Each task type carries explicit DAG edges (depends_on). Tasks
whose
748         predecessors are all already COMPLETE start as AVAILABLE; others
start

```



```

749         as UNAVAILABLE and are unlocked when their predecessors complete.
750         """
751         grid_size = self.environment['grid_size']
752         tp = self.dataset_config.get_task_params(self.scenario)
753
754         # Draw self.num_tasks entries from the typed pool (with
replacement).
755         type_pool: List[Dict] = SCENARIO_TASK_TYPES.get(self.scenario, [])
756         if not type_pool:
757             # Fallback to generic types if scenario not in table
758             type_pool = [{"id": "T0", "name": "generic", "depends_on":
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778

```

```

779         'payload_weight':
np.random.uniform(*tp['payload_weight_range']),
780         'deadline': np.random.uniform(*tp['deadline_range']),
781         # Structural depends_on uses same-episode type-id strings;
782         # resolved to int task ids after all tasks are assigned.
783         '_raw_depends_on': ttype['depends_on'],
784         'dependencies': [],
785         'location': location,
786         'urgency': np.random.uniform(0.5, 1.0),
787         'status': TASK_STATUS_AVAILABLE,
788         'assigned_robot': None,
789         'estimated_duration':
np.random.uniform(*tp['duration_range']),
790         'execution_time': 0.0,
791     }
792     tasks.append(task)
793
794     # Second pass: resolve type-level depends_on → integer task-id
dependencies.
795     # For each task, find all earlier tasks with a matching type_id.
796     # Only add dependencies when use_dependencies is enabled.
797     type_id_to_task_ids: Dict[str, List[int]] = {}
798     for t in tasks:
799         type_id_to_task_ids.setdefault(t['type_id'],
800         []).append(t['id'])
801
802     for t in tasks:
803         if self.use_dependencies:
804             dep_ids: List[int] = []
805             for dep_type in t['_raw_depends_on']:
806                 candidates = [
807                     tid for tid in type_id_to_task_ids.get(dep_type,
808                     [])
809                     if tid < t['id']]

```

```

808         ]
809         if candidates:
810             dep_ids.append(candidates[-1]) # nearest
preceding instance
811         t['dependencies'] = dep_ids
812         if dep_ids:
813             t['status'] = TASK_STATUS_UNAVAILABLE
814         del t['_raw_depends_on']
815
816     return tasks
817
818 def _generate_dynamic_obstacles(self):
819     grid_size = self.environment['grid_size']
820
821     for _ in range(sim_config.NUM_DYNAMIC_OBSTACLES):
822         position = np.array([
823             np.random.uniform(-grid_size[0]/2, grid_size[0]/2),
824             np.random.uniform(-grid_size[1]/2, grid_size[1]/2)
825         ])
826
827         speed =
np.random.uniform(*sim_config.DYNAMIC_OBSTACLE_SPEED_RANGE)
828         angle = np.random.uniform(0, 2 * np.pi)
829         velocity = speed * np.array([np.cos(angle), np.sin(angle)])
830
831         radius = np.random.uniform(0.5, 2.0)
832
833         self.dynamic_obstacles.append(DynamicObstacle(position,
velocity, radius))
834
835 def _get_observation(self) -> np.ndarray:
836     """Get flat observation with scenario-specific normalisation."""
837     obs = []
838     rp = self.robot_params

```

```

839     tp = self.dataset_config.get_task_params(self.scenario)
840     grid_half = self.local_obs_extractor.grid_half
841     max_speed = rp['max_speed']
842     battery_cap = rp['battery_capacity']
843     max_payload = rp['max_payload']
844     max_task_payload = tp['payload_weight_range'][1]
845     max_deadline = tp['deadline_range'][1]
846
847     # Robot features (max_robots × 15)
848     for i in range(self.max_robots):
849         if i < self.num_robots:
850             r = self.robots[i]
851             features = [
852                 r['position'][0] / grid_half,
853                 r['position'][1] / grid_half,
854                 r['velocity'][0] / max_speed,
855                 r['velocity'][1] / max_speed,
856                 r['max_speed'] / max_speed,
857                 r['max_payload'] / max(max_payload, 1.0),
858                 r['battery_capacity'] / max(battery_cap, 1.0),
859                 r['current_battery'] / max(battery_cap, 1.0),
860                 r['current_payload'] / max(max_payload, 1.0),
861                 r['task_completion_rate'],
862                 1.0 if r['operational_status'] == 'active' else 0.0,
863                 self.robot_loads.get(i, 0) / max(self.max_robots, 1),
864                 1.0 if r['current_task'] is not None else 0.0,
865                 0.0, 0.0
866             ]
867         else:
868             features = [0.0] * self.robot_feature_dim
869         obs.extend(features)
870
871     # Task features (max_tasks × 12)
872     for i in range(self.max_tasks):

```

```

873         if i < self.num_tasks:
874             t = self.tasks[i]
875             features = [
876                 t['location'][0] / grid_half,
877                 t['location'][1] / grid_half,
878                 self.task_types.get(t['type'], 0) /
max(len(self.task_types), 1),
879                 self.priority_encoding.get(t['priority'], 1) / 3.0,
880                 t['payload_weight'] / max(max_task_payload, 1.0),
881                 t['deadline'] / max(max_deadline, 1.0),
882                 len(t['dependencies']) / 5.0,
883                 1.0 if i in self.completed_tasks else 0.0,
884                 1.0 if i in self.task_assignments else 0.0,
885                 t['urgency'],
886                 self.task_progress.get(i, 0.0),
887                 0.0
888             ]
889         else:
890             features = [0.0] * self.task_feature_dim
891             obs.extend(features)
892
893         # Global features (10)
894         global_features = [
895             len(self.completed_tasks) / max(self.num_tasks, 1),
896             self.current_step / max(self._max_steps, 1),
897             len(self.task_assignments) / max(self.num_tasks, 1),
898             self.num_robots / max(self.max_robots, 1),
899             self.num_tasks / max(self.max_tasks, 1),
900             sum(r['current_battery'] for r in self.robots)
901                 / max(self.num_robots * battery_cap, 1.0),
902             self.collisions / 100.0,
903             len(self.dynamic_obstacles) / 10.0,
904             self.robot_collisions / 100.0,
905             # Slot 9: maritime current strength (normalised 0-1); 0 for

```

```

other scenarios
906         float(self._maritime_current_strength) / 3.0,
907     ]
908     obs.extend(global_features)
909
910     return np.array(obs, dtype=np.float32)
911
912 def _get_info(self) -> Dict:
913     n_completed = len(self.completed_tasks)
914     energy_efficiency = n_completed / max(self.total_energy, 1e-6)
915     energy_per_task    = self.total_energy / max(n_completed, 1)
916
917     # DCMRTA improvement D: task_waiting_efficiency
918     # = mean sum_waiting_time per task (lower = better coordination)
919     total_wait = sum(self._task_wait_times.get(t.get('id', i), 0.0)
920                     for i, t in enumerate(getattr(self, 'tasks',
921 [])))
922     task_waiting_efficiency = total_wait /
923 max(len(self._task_wait_times), 1)
924
925     # DCMRTA improvement D: makespan = current_step if all done, else
926 inf
927     makespan = (float(self.current_step)
928                 if len(self.completed_tasks) >= self.num_tasks
929                 else float(self._max_steps))
930
931     # DCMRTA improvement A: total travel distance (normalised)
932     travel_dist = sum(self._robot_travel_dist.values())
933
934     # DCMRTA improvement B: mean waiting time per robot
935     waiting_time = (sum(self._robot_waiting_time.values()) /
936                     max(len(self._robot_waiting_time), 1))
937
938     # DCMRTA improvement C: abandoned agent count

```

```

936         abandoned_count = len(self._abandoned_agents)
937
938         # time_cost: mean step at which tasks were completed
939         time_cost = (float(self.current_step) / max(n_completed, 1))
940
941         return {
942             'completion_rate':          n_completed / max(1,
self.num_tasks),
943             'assignment_rate':          len(self.task_assignments) /
max(1, self.num_tasks),
944             'total_energy':             self.total_energy,
945             'energy_efficiency':        energy_efficiency,
946             'energy_per_task':          energy_per_task,
947             'collisions':               self.collisions,
948             'robot_collisions':         self.robot_collisions,
949             'reallocation_count':       self.reallocation_count,
950             'reallocation_events':      self.reallocation_events,
951             'step':                     self.current_step,
952             'num_robots':               self.num_robots,
953             'num_tasks':               self.num_tasks,
954             # DCMRTA metrics (D, A, B, C)
955             'task_waiting_efficiency':  task_waiting_efficiency,
956             'makespan':                 makespan,
957             'travel_dist':              travel_dist,
958             'waiting_time':             waiting_time,
959             'time_cost':                time_cost,
960             'abandoned_agent_count':   abandoned_count,
961         }
962
963     def _update_robot_positions(self, dt: float):
964         """Update robot positions (realistic execution). Tracks
travel_dist and waiting_time."""
965         for i, robot in enumerate(self.robots):
966             if robot.get('operational_status') != ROBOT_STATUS_ACTIVE:

```

```

967         robot['velocity'] = np.zeros(2)
968         # DCMRTA B: robot waiting while inactive
969         self._robot_waiting_time[i] =
self._robot_waiting_time.get(i, 0.0) + dt
970         continue
971
972         target_task_id = self.robot_targets[i]
973
974         # Variable-robot curriculum can shrink self.tasks mid-episode;
975         # a stale target_task_id would raise IndexError on the next
line.
976         if (target_task_id is None
977             or target_task_id >= len(self.tasks)
978             or target_task_id < 0):
979             robot['velocity'] = np.zeros(2)
980             self._robot_waiting_time[i] =
self._robot_waiting_time.get(i, 0.0) + dt
981             self.robot_targets[i] = None
982             robot['current_task'] = None
983             continue
984
985         task = self.tasks[target_task_id]
986         target_pos = np.array(task['location'])
987         current_pos = np.array(robot['position'])
988
989         direction = target_pos - current_pos
990         distance = np.linalg.norm(direction)
991         # DCMRTA improvement A: accumulate travel distance per robot
992         prev = self._robot_prev_pos.get(i)
993         if prev is not None:
994             step_dist = np.linalg.norm(np.array(robot['position']) -
np.array(prev))
995             self._robot_travel_dist[i] =
self._robot_travel_dist.get(i, 0.0) + float(step_dist)

```



```

996         self._robot_prev_pos[i] = list(robot['position'])
997
998         if distance > self._arrival_tol: # was hard-coded 0.5, now
scenario-scaled
999             # use per-robot max_speed_loaded (scenario-specific, ±15%
noise)
1000             # instead of the global SPEED_LOADED_FACTOR (always 0.6).
1001             # Unloaded robot['max_speed']; loaded
robot['max_speed_loaded'].
1002             if robot['current_payload'] > 0:
1003                 speed = robot['max_speed_loaded']
1004             else:
1005                 speed = robot['max_speed']
1006
1007             velocity = direction / distance * min(speed, distance /
1008             dt)
1009
1010             # Potential-field repulsion: steer away from nearby
robots and
1011             # obstacles so robots naturally avoid collisions during
movement.
1012             # Applied as a velocity correction before position
integration.
1013             _repulsion = np.zeros(2)
1014             _rep_robot_radius = 2.0 # start repelling at this
inter-robot distance (m)
1015             _rep_obs_margin = 1.0 # safety margin beyond obstacle
radius (m)
1016             _rep_strength = 0.35 # repulsion magnitude as
fraction of speed
1017
1018             for _j, _other in enumerate(self.robots):
1019                 if _j == i:
1020                     continue

```

```

1020         if _other.get('operational_status') !=
ROBOT_STATUS_ACTIVE:
1021             continue
1022             _diff = current_pos - np.array(_other['position'])
1023             _d = np.linalg.norm(_diff)
1024             if 0.0 < _d < _rep_robot_radius:
1025                 _repulsion += (_diff / _d) * _rep_strength *
speed * (
1026                     1.0 - _d / _rep_robot_radius)
1027
1028             _obstacles = self.environment.get('obstacles',
np.empty((0, 3)))
1029             if isinstance(_obstacles, np.ndarray) and _obstacles.ndim
== 2:
1030                 for _obs in _obstacles:
1031                     _obs_pos = _obs[:2]
1032                     _obs_r = float(_obs[2]) if len(_obs) > 2 else
0.5
1033                     _diff = current_pos - _obs_pos
1034                     _d = np.linalg.norm(_diff)
1035                     _margin = _obs_r + _rep_obs_margin
1036                     if 0.0 < _d < _margin:
1037                         _repulsion += (_diff / _d) * _rep_strength *
speed * (
1038                             1.0 - _d / _margin)
1039
1040                 velocity = velocity + _repulsion
1041                 _v_norm = np.linalg.norm(velocity)
1042                 if _v_norm > speed:
1043                     velocity = velocity / _v_norm * speed
1044
1045                 # Maritime physics: ocean current drift + resistance
1046                 # Current pushes the robot off-course; the robot must
exped

```

```

1047         # extra energy fighting the drift to stay on track.
1048         # Wave height adds hull-resistance drag (quadratic in
height).
1049         # Wind adds a small surface-drift component.
1050         if self.scenario == 'maritime_security':
1051             cs = self._maritime_current_strength # m/s
equivalent
1052             wh = self._maritime_wave_height # metres
1053             ws = self._maritime_wind_speed # m/s
1054             cd = self._maritime_current_dir # unit vector
1055
1056             # 1. Ocean-current drift displaces position
1057             current_drift = cd * cs * dt
1058             # 2. Wind adds a small surface drift (5% of wind
speed)
1059             wind_dir = cd * np.array([-cd[1], cd[0]]) #
perpendicular
1060             wind_drift = wind_dir * (ws * 0.05) * dt
1061
1062             # 3. Robot must compensate - effective speed is
reduced
1063             # by the component of current opposing motion.
1064             opposing = -np.dot(cd, direction / (distance +
1e-8))
1065             speed_loss = cs * max(0.0, opposing)
1066             eff_speed = max(0.1, speed - speed_loss)
1067
1068             velocity = direction / distance * min(eff_speed,
distance / dt)
1069             new_pos = current_pos + velocity * dt +
current_drift + wind_drift
1070
1071             # Clamp to grid bounds
1072             gs = self.environment['grid_size']

```

```

1073         new_pos[0] = np.clip(new_pos[0], -gs[0] / 2, gs[0] /
1074         2)
1075         new_pos[1] = np.clip(new_pos[1], -gs[1] / 2, gs[1] /
1076         2)
1077         robot['position'] = new_pos
1078     else:
1079         robot['position'] = current_pos + velocity * dt
1080
1081         robot['velocity'] = velocity
1082
1083         # Battery drain
1084         # multiply drain by battery_drain_loaded_factor when
1085         carrying
1086         # payload (warehouse +40%, SAR +90%, maritime +70%).
1087         actual_speed = np.linalg.norm(velocity)
1088         base_drain = robot.get('battery_drain_rate',
1089
1090         self.dataset_config.ROBOT_BATTERY_DRAIN_RATE)
1091         if robot['current_payload'] > 0:
1092             base_drain *=
1093             robot.get('battery_drain_loaded_factor', 1.0)
1094             drain = (base_drain
1095                     + robot.get('battery_drain_speed_coef', 0.012) *
1096             actual_speed ** 2
1097
1098                     ) * dt
1099
1100             # Maritime: extra drain from wave resistance and
1101             # energy spent fighting the current.
1102             if self.scenario == 'maritime_security':
1103                 wave_drag_coef = 0.008
1104                 current_work_coef = 0.015
1105                 drain += (wave_drag_coef *
1106                         (self._maritime_wave_height ** 2)
1107
1108                         + current_work_coef *

```

```

self._maritime_current_strength) * dt
1100
1101         robot['current_battery'] -= drain
1102
1103         # accumulate energy consumed (convert drain-rate dt to Wh)
1104         self.total_energy += drain
1105
1106         # deactivate robot when battery exhausted
1107         if robot['current_battery'] <= 0.0:
1108             robot['current_battery'] = 0.0
1109             robot['operational_status'] = ROBOT_STATUS_INACTIVE
1110             robot['velocity'] = np.zeros(2)
1111             # Release any assigned task back to AVAILABLE
1112             if target_task_id is not None and target_task_id not
in self.completed_tasks:
1113                 self.tasks[target_task_id]['status'] =
TASK_STATUS_AVAILABLE
1114                 self.tasks[target_task_id]['assigned_robot'] =
None
1115                 if target_task_id in self.task_assignments:
1116                     del self.task_assignments[target_task_id]
1117                     self.robot_loads[i] = 0
1118                     robot['current_payload'] = max(
1119                         0, robot['current_payload'] -
task['payload_weight'])
1120                     robot['current_task'] = None
1121                     self.robot_targets[i] = None
1122                     # Dynamic reallocation: robot failure
1123                     self._trigger_reallocation(
1124                         task_id=target_task_id,
1125                         reason='robot_failure',
1126                         failed_robot_id=i
1127                     )
1128         else:

```

```

1129         # At target - execute task
1130         robot['velocity'] = np.zeros(2)
1131         self.task_progress[target_task_id] += dt /
task['estimated_duration']
1132
1133         if self.task_progress[target_task_id] >= 1.0:
1134             # Task completed
1135             self._complete_task(target_task_id, i)
1136
1137     def _complete_task(self, task_id: int, robot_id: int):
1138         """Mark task COMPLETE, unlock dependents, and auto-reassign the
robot.
1139
1140         Applies  $\times$ DEPENDENT_TASK_REWARD_MULTIPLIER to the completion
reward for
1141         tasks that had non-empty dependencies (they required prior work
to unlock).
1142         Grants an energy-efficiency bonus proportional to remaining
battery.
1143         """
1144         if task_id in self.completed_tasks:
1145             return
1146
1147         self.completed_tasks.add(task_id)
1148         task = self.tasks[task_id]
1149         task['status'] = TASK_STATUS_COMPLETE
1150
1151         robot = self.robots[robot_id]
1152
1153         robot['current_payload'] = max(0, robot['current_payload'] -
task['payload_weight'])
1154         robot['current_task'] = None
1155         self.robot_targets[robot_id] = None
1156         self.robot_loads[robot_id] = max(0, self.robot_loads[robot_id]

```

- 1)

```
1157
1158     # EMA of observed completion success ( $\alpha = 0.1$ )
1159     robot['task_completion_rate'] = (
1160         0.9 * robot['task_completion_rate'] + 0.1 * 1.0
1161     )
1162
1163     #  $\times 1.5$  reward multiplier for dependent tasks
1164     if task.get('dependencies'):
1165         self._pending_efficiency_reward = getattr(
1166             self, '_pending_efficiency_reward', 0.0)
1167         self._pending_efficiency_reward += (
1168             self.config.REWARD_TASK_COMPLETION *
1169             (DEPENDENT_TASK_REWARD_MULTIPLIER - 1.0)
1170         )
1171
1172     # Unlock tasks whose dependencies are now all COMPLETE
1173     for t in self.tasks:
1174         if t['status'] == TASK_STATUS_UNAVAILABLE:
1175             if all(d in self.completed_tasks for d in
1176                 t['dependencies']):
1177                 t['status'] = TASK_STATUS_AVAILABLE
1178
1179     # Auto-reassign: find next-nearest AVAILABLE task for this robot.
1180     # Skipped when ScenarioConfig.auto_reassign_on_complete=False
1181     (warehouse):
1182     # the RL policy is then the sole arbiter of all assignments,
1183     which gives
1184     # Q-values a meaningful gradient instead of converging to a
1185     constant.
1186     _sc_cfg = SCENARIO_CONFIGS.get(self.scenario)
1187     if _sc_cfg is None or _sc_cfg.auto_reassign_on_complete:
1188         rpos = np.array(robot['position'])
1189         best_tid, best_d = None, float('inf')
```

```

1185         for tid in range(self.num_tasks):
1186             if self.tasks[tid]['status'] != TASK_STATUS_AVAILABLE:
1187                 continue
1188             if tid in self.task_assignments:
1189                 continue
1190             d = float(np.linalg.norm(rpos -
np.array(self.tasks[tid]['location'])))
1191             if d < best_d:
1192                 best_d, best_tid = d, tid
1193             if best_tid is not None:
1194                 self.robot_targets[robot_id] = best_tid
1195                 self.task_assignments[best_tid] = robot_id
1196                 self.robot_loads[robot_id] = 1
1197                 robot['current_task'] = best_tid
1198                 self.tasks[best_tid]['status'] =
TASK_STATUS_IN_PROGRESS
1199                 self.tasks[best_tid]['assigned_robot'] = robot_id
1200
1201             # Energy-efficiency bonus --- rewards frugal battery use
1202             batt_cap = max(robot.get('battery_capacity', 100.0), 1.0)
1203             efficiency_ratio = robot['current_battery'] / batt_cap
1204             self._pending_efficiency_reward = getattr(self,
'_pending_efficiency_reward', 0.0)
1205             self._pending_efficiency_reward += self.config.REWARD_EFFICIENCY
* efficiency_ratio
1206
1207     def _check_collisions(self) -> Tuple[int, int]:
1208         """Check for collisions"""
1209         obstacle_collisions = 0
1210         robot_robot_collisions = 0
1211
1212         obstacles = self.environment.get('obstacles', np.array([]))
1213
1214         for robot in self.robots:

```



```

1215         pos = np.array(robot['position'])
1216
1217         # Static obstacles
1218         if len(obstacles) > 0:
1219             for obs in obstacles:
1220                 dist = np.linalg.norm(pos - obs[:2])
1221                 if dist < robot['radius'] + obs[2]:
1222                     obstacle_collisions += 1
1223
1224         # Dynamic obstacles
1225         for dyn_obs in self.dynamic_obstacles:
1226             dist = np.linalg.norm(pos - dyn_obs.position)
1227             if dist < robot['radius'] + dyn_obs.radius:
1228                 obstacle_collisions += 1
1229
1230         # Robot-robot collisions
1231         for i in range(self.num_robots):
1232             for j in range(i + 1, self.num_robots):
1233                 pos_i = np.array(self.robots[i]['position'])
1234                 pos_j = np.array(self.robots[j]['position'])
1235                 dist = np.linalg.norm(pos_i - pos_j)
1236                 min_dist = self.robots[i]['radius'] +
self.robots[j]['radius']
1237                 if dist < min_dist:
1238                     robot_robot_collisions += 1
1239
1240         return obstacle_collisions, robot_robot_collisions
1241
1242     def _trigger_reallocation(self, task_id: int, reason: str,
1243                             failed_robot_id: int = None):
1244         """
1245         Attempt to reassign a released task to the best available active robot.
1246
1247         Selection criteria (greedy priority):

```

```

1248     1. Robot must be active and have sufficient battery & payload capacity.
1249     2. All task dependencies must be satisfied.
1250     3. Prefer: highest remaining battery, then shortest distance to task.
1251
1252     Records the event in self.reallocation_events regardless of success.
1253     """
1254
1255     task = self.tasks[task_id]
1256     if task_id in self.completed_tasks:
1257         return # already done - nothing to reassign
1258
1259     candidates = []
1260     task_loc = np.array(task['location'])
1261
1262     for r_id, robot in enumerate(self.robots):
1263         if r_id == failed_robot_id:
1264             continue
1265         if robot.get('operational_status') != ROBOT_STATUS_ACTIVE:
1266             continue
1267         # Capacity check
1268         if robot['current_payload'] + task['payload_weight'] >
1269         robot['max_payload']:
1270             continue
1271         # Battery sanity: must have at least 5% remaining
1272         if robot['current_battery'] < robot['battery_capacity'] *
1273         0.05:
1274             continue
1275         # Dependencies check
1276         if not all(d in self.completed_tasks for d in
1277         task['dependencies']):
1278             continue
1279
1280         dist = float(np.linalg.norm(np.array(robot['position']) -
1281         task_loc))
1282         # Score: prioritise battery life, penalise distance

```

```

1278         score = robot['current_battery'] - 0.5 * dist
1279         candidates.append((score, r_id))
1280
1281     event = {
1282         'step':            self.current_step,
1283         'task_id':         task_id,
1284         'reason':          reason,
1285         'failed_robot_id': failed_robot_id,
1286         'new_robot_id':    None,
1287         'success':         False,
1288     }
1289
1290     if candidates:
1291         candidates.sort(key=lambda x: -x[0])
1292         _, best_robot_id = candidates[0]
1293         best_robot = self.robots[best_robot_id]
1294
1295         # Assign the task
1296         self.task_assignments[task_id]          = best_robot_id
1297         self.robot_loads[best_robot_id]          += 1
1298         best_robot['current_payload']            +=
1299         task['payload_weight']
1300         best_robot['current_task']                = task_id
1301         task['status']                            =
1302         TASK_STATUS_IN_PROGRESS
1303         task['assigned_robot']                    = best_robot_id
1304         self.robot_targets[best_robot_id]        = task_id
1305
1306         event['new_robot_id'] = best_robot_id
1307         event['success']      = True
1308         self.reallocation_count += 1
1309         logger.debug(
1310             f"Reallocation [{reason}]: task {task_id} → robot
1311             {best_robot_id} "

```

```

1309         f"(step {self.current_step})"
1310     )
1311
1312     self.reallocation_events.append(event)
1313
1314     def _check_and_reallocate(self):
1315         """
1316         Proactive mid-episode reallocation sweep called every step.
1317         Includes DCMRTA improvement B: coalition patience timeout for all
1318             scenarios.
1319
1320         Triggers reallocation for:
1321         a) Deadline-critical unassigned tasks (< 15% deadline remaining).
1322         b) Load imbalance: idle robots exist while some robots carry  $\geq 2$  tasks.
1323         """
1324         if self._last_realloc_step == self.current_step:
1325             return # Already ran this step (called twice via robot-death
1326                 path)
1327
1328         self._last_realloc_step = self.current_step
1329
1330         remaining_time = self._max_steps - self.current_step
1331         dt = self.dataset_config.DT
1332
1333         # (a) Deadline-critical unassigned tasks
1334         for task in self.tasks:
1335             t_id = task['id']
1336             if t_id in self.completed_tasks or t_id in
1337                 self.task_assignments:
1338                 continue # done or already covered
1339             if task['status'] == TASK_STATUS_AVAILABLE:
1340                 # Estimate deadline urgency using steps remaining
1341                 deadline_steps = task['deadline'] / max(dt, 1e-6)
1342                 if deadline_steps > 0 and remaining_time / deadline_steps
1343                     < 0.15:

```

```

1339         self._trigger_reallocation(
1340             task_id=t_id,
1341             reason='deadline_urgency'
1342         )
1343
1344         # DCMRTA improvement B: Coalition patience timeout (all
scenarios)
1345         # If a robot has been waiting for a coalition partner longer than
1346         # COALITION_MAX_WAITING_TIME steps, abandon and reassign (all
scenarios)
1347         for robot_id, wait_start in
list(self._coalition_wait_start.items()):
1348             if self.current_step - wait_start >
COALITION_MAX_WAITING_TIME:
1349                 # Find which task this robot is waiting on
1350                 waiting_task = next(
1351                     (tid for tid, rid in self.task_assignments.items()
1352                      if rid == robot_id and tid not in
self.completed_tasks), None)
1353                 if waiting_task is not None:
1354                     self._abandoned_agents.append((robot_id,
waiting_task, 'coalition_timeout'))
1355                     # Record waiting time for this task
1356                     self._task_wait_times[waiting_task] = (
1357                         self._task_wait_times.get(waiting_task, 0.0) +
1358                         float(self.current_step - wait_start))
1359                     self._trigger_reallocation(waiting_task,
'coalition_timeout',
1360                                             failed_robot_id=robot_id)
1361                     del self._coalition_wait_start[robot_id]
1362
1363         # TA-RWARE improvement K: StuckCounter for all scenarios
1364         # A robot dwelling at its target position (within arrival_tol)
has an

```

```

1365         # integer-rounded position that stops changing, which would
1366         trigger a
1367         # false-positive stuck detection. The dwelling check below
1368         prevents
1369         # evicting the target while task_progress is still accumulating.
1370         if not hasattr(self, '_stuck_counters'):
1371             self._stuck_counters = {}
1372         for i, robot in enumerate(self.robots):
1373             pos = tuple(int(p) for p in robot.get('position', [0, 0]))
1374             if i not in self._stuck_counters:
1375                 self._stuck_counters[i] = StuckCounter(pos)
1376
1377         # Robots accumulating task_progress at their target are not
1378         stuck;
1379         # evicting their target would halt progress counting.
1380         tid_now = self.robot_targets.get(i)
1381         is_dwelling = False
1382         if tid_now is not None and 0 <= tid_now < len(self.tasks):
1383             _tloc = self.tasks[tid_now].get('location')
1384             if _tloc is not None:
1385                 _d = float(np.linalg.norm(
1386                     np.array(robot.get('position', [0, 0]))
1387                     - np.array(_tloc)))
1388                 if _d <= self._arrival_tol:
1389                     is_dwelling = True
1390
1391         if is_dwelling:
1392             self._stuck_counters[i] = StuckCounter(pos)
1393             continue
1394
1395         status = self._stuck_counters[i].update(pos)
1396         if status == 'replan' and i in self.robot_targets and
1397         self.robot_targets[i] is not None:
1398             tid = self.robot_targets[i]
1399             if tid not in self.completed_tasks:

```

```

1395         self._trigger_reallocation(tid, 'stuck_replan',
failed_robot_id=i)
1396     elif status == 'free':
1397         robot['current_task'] = None
1398         self.robot_targets[i] = None
1399         self._stuck_counters[i].reset(pos)
1400
1401     # (b-pre) Proactive idle-robot assignment sweep
1402     # Assign every idle active robot to the nearest available pending
task.
1403     # This covers two cases the existing auto-assignment misses:
1404     # 1. Initially idle robots when num_tasks < num_robots
1405     # 2. Robots whose dependency-unlocked tasks weren't claimed at
completion
1406     # Runs BEFORE the load-imbalance check so the overloaded check
sees
1407     # an accurate picture of how many robots are truly unassigned.
1408     # Skipped for scenarios where auto_reassign_on_complete=False so
that
1409     # the RL policy is the sole arbiter of all assignments.
1410     _sc_cfg_realloc = SCENARIO_CONFIGS.get(self.scenario)
1411     _auto_realloc = (_sc_cfg_realloc is None
1412                     or _sc_cfg_realloc.auto_reassign_on_complete)
1413     if _auto_realloc:
1414         for robot_id, robot in enumerate(self.robots):
1415             if robot.get('operational_status') != ROBOT_STATUS_ACTIVE:
1416                 continue
1417             if self.robot_targets.get(robot_id) is not None:
1418                 continue # already has a target
1419             rpos = np.array(robot['position'])
1420             best_tid2, best_d2 = None, float('inf')
1421             for tid in range(self.num_tasks):
1422                 if self.tasks[tid]['status'] != TASK_STATUS_AVAILABLE:
1423                     continue

```

```

1424         if tid in self.task_assignments:
1425             continue
1426         d2 = float(np.linalg.norm(rpos -
np.array(self.tasks[tid]['location'])))
1427         if d2 < best_d2:
1428             best_d2, best_tid2 = d2, tid
1429         if best_tid2 is not None:
1430             self.robot_targets[robot_id] = best_tid2
1431             self.task_assignments[best_tid2] = robot_id
1432             self.robot_loads[robot_id] = 1
1433             robot['current_task'] = best_tid2
1434             self.tasks[best_tid2]['status'] =
TASK_STATUS_IN_PROGRESS
1435             self.tasks[best_tid2]['assigned_robot'] = robot_id
1436
1437         # (b) Load imbalance reallocation
1438         active_robots = [r for r in self.robots if
r.get('operational_status') == ROBOT_STATUS_ACTIVE]
1439         idle_robots = [r for r in active_robots if r['current_task']
is None]
1440         overloaded_ids = [
1441             r['id'] for r in active_robots
1442             if self.robot_loads.get(r['id'], 0) >= 2
1443         ]
1444
1445         if idle_robots and overloaded_ids:
1446             # Find pending tasks whose assigned robot is overloaded
1447             for task in self.tasks:
1448                 t_id = task['id']
1449                 if t_id in self.completed_tasks:
1450                     continue
1451                 if task.get('assigned_robot') in overloaded_ids:
1452                     # Only rebalance low-priority tasks to avoid
disrupting urgent work

```



```

1453         if task['priority'] == 'low':
1454             # Release from overloaded robot
1455             old_robot_id = task['assigned_robot']
1456             old_robot = self.robots[old_robot_id]
1457             old_robot['current_payload'] = max(
1458                 0, old_robot['current_payload'] -
task['payload_weight'])
1459             old_robot['current_task'] = None
1460             self.robot_targets[old_robot_id] = None
1461             self.robot_loads[old_robot_id] = max(
1462                 0, self.robot_loads[old_robot_id] - 1)
1463             if t_id in self.task_assignments:
1464                 del self.task_assignments[t_id]
1465             task['status'] = TASK_STATUS_AVAILABLE
1466             task['assigned_robot'] = None
1467
1468             self._trigger_reallocation(
1469                 task_id=t_id,
1470                 reason='load_imbalance'
1471             )
1472             break # one rebalance per step to avoid thrashing
1473
1474     def step(self, action: int) -> Tuple[np.ndarray, float, bool, bool,
Dict]:
1475         reward = 0.0
1476         info = {'task_completed': False, 'task_assigned': False,
'collision': False}
1477
1478         dt = self.dataset_config.DT
1479
1480         # Update dynamic obstacles
1481         if self.dynamic_obstacles:
1482             bounds = (
1483                 -self.environment['grid_size'][0]/2,

```

```

1484         self.environment['grid_size'][0]/2,
1485         -self.environment['grid_size'][1]/2,
1486         self.environment['grid_size'][1]/2
1487     )
1488     for obs in self.dynamic_obstacles:
1489         obs.update(dt, bounds)
1490
1491     # Update robot positions (realistic task execution)
1492     if sim_config.TASK_TRAVEL_REQUIRED:
1493         self._update_robot_positions(dt)
1494
1495     # Proactive dynamic task reallocation sweep
1496     self._check_and_reallocate()
1497
1498     # Process action
1499     _sc_step = SCENARIO_CONFIGS.get(self.scenario)
1500     _rl_only = _sc_step is not None and not
1501     _sc_step.auto_reassign_on_complete
1502     if action == self.max_robots * self.max_tasks:
1503         if not _rl_only:
1504             reward -= 0.1 # SAR/maritime: always penalize NOOP
1505         else:
1506             # Warehouse (RL-only): only penalize when an assignment
1507             was possible.
1508             # When all robots are busy traveling, NOOP is the only
1509             legal action;
1510             # charging -0.1 in that case floods the replay buffer
1511             with spurious
1512             # negatives and destabilizes DQN/MAPPO/HybridRL after IL
1513             fades out.
1514     _can_assign = (
1515         any(self.robot_targets.get(r) is None
1516            for r in range(self.num_robots)) and
1517         any(t['status'] == TASK_STATUS_AVAILABLE for t in

```

```

self.tasks)
1513         )
1514         if _can_assign:
1515             reward -= 0.1
1516         else:
1517             robot_id = action // self.max_tasks
1518             task_id = action % self.max_tasks
1519
1520             # In RL-only mode (e.g. warehouse), reject assignments to
busy robots.
1521             # Without this guard the RL overwrites robot_targets while
the old task
1522             # remains IN_PROGRESS forever, causing task-state corruption
that shrinks
1523             # the pool of available tasks and kills the learning signal.
1524             _robot_busy = _rl_only and self.robot_targets.get(robot_id)
is not None
1525
1526             if (not _robot_busy and
1527                 robot_id < self.num_robots and task_id <
self.num_tasks and
1528                 self.tasks[task_id]['status'] ==
TASK_STATUS_AVAILABLE):
1529
1530                 robot = self.robots[robot_id]
1531                 task = self.tasks[task_id]
1532
1533                 # Check payload capacity
1534                 if robot['current_payload'] + task['payload_weight'] <=
robot['max_payload']:
1535
1536                     # Assign task --- transition AVAILABLE → IN_PROGRESS
1537                     self.task_assignments[task_id] = robot_id
1538                     self.robot_loads[robot_id] += 1

```

```

1539         robot['current_payload'] += task['payload_weight']
1540         robot['current_task'] = task_id
1541         task['status'] = TASK_STATUS_IN_PROGRESS
1542         task['assigned_robot'] = robot_id
1543         self.robot_targets[robot_id] = task_id
1544         info['task_assigned'] = True
1545
1546         # Priority bonus
1547         priority_bonus =
self.priority_encoding.get(task['priority'], 1) / 3.0
1548         reward += self.config.REWARD_PRIORITY_BONUS *
priority_bonus
1549     else:
1550         reward += self.config.PENALTY_INCOMPLETE * 0.5
1551         # EMA update toward 0 on constraint-violation failure
1552         robot = self.robots[robot_id]
1553         robot['task_completion_rate'] = (
1554             0.9 * robot['task_completion_rate'] + 0.1 * 0.0
1555         )
1556     else:
1557         reward += self.config.PENALTY_INCOMPLETE
1558
1559     # Check collisions
1560     obs_col, rob_col = self._check_collisions()
1561     self.collisions += obs_col
1562     self.robot_collisions += rob_col
1563
1564     if obs_col > 0:
1565         reward += self.config.PENALTY_COLLISION
1566         info['collision'] = True
1567     if rob_col > 0:
1568         reward += self.config.PENALTY_ROBOT_COLLISION
1569
1570     # Fix 5: collect any energy-efficiency bonus earned by

```

```

1571         _complete_task this step
        if hasattr(self, '_pending_efficiency_reward') and
        self._pending_efficiency_reward:
1572             reward += self._pending_efficiency_reward
1573             self._pending_efficiency_reward = 0.0
1574
1575             # Issue reward for tasks that finished travel execution this step.
1576             # task_assignments persists for the episode lifetime, so
            _rewarded_task_ids
1577             # gates this loop to fire exactly once per completion event.
1578             # Iterating all keys (not just new completions) correctly rewards
1579             # simultaneous arrivals from multiple robots in the same step.
            for task_id in list(self.task_assignments.keys()):
1580                 if (task_id in self.completed_tasks
1581                     and task_id not in self._rewarded_task_ids):
1582                     self._rewarded_task_ids.add(task_id)
1583                     info['task_completed'] = True
1584                     reward += self.config.REWARD_TASK_COMPLETION
1585
1586
1587             self.current_step += 1
1588
1589             terminated = len(self.completed_tasks) == self.num_tasks
1590             truncated = self.current_step >= self._max_steps
1591
1592             if terminated:
1593                 reward += self.config.REWARD_TASK_COMPLETION * 2
1594
1595             info.update(self._get_info())
1596
1597             return self._get_observation(), reward, terminated, truncated,
            info
1598
1599         def record_episode_outcome(self, completion_rate: float):
1600             """Forward episode outcome to whichever difficulty manager is

```

```

1600         active."""
1601         if self.performance_generator is not None:
1602             self.performance_generator.record(completion_rate)
1603         if self.curriculum is not None:
1604             self.curriculum.record_performance(completion_rate)
1605             self.curriculum.advance()
1606
1607     def get_state_dim(self) -> int:
1608         return self.observation_space.shape[0]
1609
1610     def get_action_dim(self) -> int:
1611         return self.action_space.n
1612
1613     def get_global_state(self) -> Dict:
1614         """Get global state for multi-agent methods.
1615
1616         Dynamic obstacles are merged into 'obstacles' so that
1617         LocalObservationExtractor and GlobalStateExtractor can include them
1618         in agent observations. Without this, agents are penalised for
1619         collisions with moving objects they literally cannot see.
1620         Each DynamicObstacle is converted to the [x, y, radius] format
1621         that the extractor and collision checker already use for static
1622         obstacles.
1623         """
1624         static_obs = list(self.environment.get('obstacles', []))
1625         dynamic_obs = [
1626             np.array([obs.position[0], obs.position[1], obs.radius])
1627             for obs in self.dynamic_obstacles
1628         ]
1629         all_obstacles = static_obs + dynamic_obs
1630         return {
1631             'robots': self.robots,
1632             'tasks': self.tasks,
1633             'obstacles': all_obstacles,

```

```

1633         'current_step': self.current_step
1634     }
1635
1636     def get_action_mask(self) -> np.ndarray:
1637         """
1638         Boolean mask of shape (action_dim,) where True = valid action.
1639         Action index = robot_id * max_tasks + task_id.
1640         Index max_robots*max_tasks is the no-op, always valid.
1641         In RL-only mode (warehouse): action valid only when task is AVAILABLE and
1642         robot is idle; this prevents wasted samples and matches the step() guard.
1643         """
1644         _sc_m = SCENARIO_CONFIGS.get(self.scenario)
1645         _rl_only_m = _sc_m is not None and not
1646             _sc_m.auto_reassign_on_complete
1647         mask = np.zeros(self.action_space.n, dtype=bool)
1648         for r in range(self.num_robots):
1649             if _rl_only_m and self.robot_targets.get(r) is not None:
1650                 continue # robot busy in RL-only mode --- no valid
1651                 assignment
1652             for t in range(self.num_tasks):
1653                 if self.tasks[t]['status'] == TASK_STATUS_AVAILABLE:
1654                     mask[r * self.max_tasks + t] = True
1655             mask[self.max_robots * self.max_tasks] = True # no-op always
1656             valid
1657         return mask
1658
1659     def get_local_observations(self) -> List[np.ndarray]:
1660         """Get local observations for all agents"""
1661         global_state = self.get_global_state()
1662         return [
1663             self.local_obs_extractor.extract(global_state, i)
1664             for i in range(self.num_robots)
1665         ]
1666

```

```

1664     def get_global_observation(self) -> np.ndarray:
1665         """Get global observation for centralized critic"""
1666         global_state = self.get_global_state()
1667         return self.global_obs_extractor.extract(global_state)
1668
1669
1670 # TRAINING AND EVALUATION PIPELINE

```

Listing B.8: Improved task allocation environment, the central Gymnasium environment class for all three scenarios.

```

1  """
2  environment/real_world_env.py
3  =====
4  Gym-compatible wrapper for real-world robot deployments.
5
6  Inherits ImprovedTaskAllocationEnv so every method that rl_trainer calls
7  (get_action_mask, _get_observation, _get_info, etc.) works unchanged.
8  The only override is reset() and step(): instead of running physics
9  simulation, they read live state from user-supplied sensor callbacks and
10 send decoded actions to a user-supplied command callback.
11
12 Required robot state dict keys (returned by robot_state_fn):
13     id, position (np.array [x,y]), velocity (np.array [x,y]),
14     current_battery, battery_capacity, battery_drain_rate,
15     battery_drain_loaded_factor, max_speed, max_speed_loaded,
16     max_payload, current_payload, operational_status, current_task,
17     radius, observation_radius, communication_range, task_completion_rate
18
19 Required task state dict keys (returned by task_state_fn):
20     id, location (np.array [x,y]), status, priority,
21     payload_weight, deadline
22
23 Use TASK_STATUS_* and ROBOT_STATUS_* constants from config.base_config

```



```

24 for the status fields.
25 """
26 from __future__ import annotations
27
28 import time
29 import logging
30 import threading
31 from typing import Callable, Dict, List, Optional, Tuple, Any
32
33 import numpy as np
34 import gymnasium as gym
35
36 from config.base_config import (
37     TASK_STATUS_COMPLETE, TASK_STATUS_AVAILABLE,
38     ROBOT_STATUS_ACTIVE, ROBOT_STATUS_FAILED,
39     LOCAL_OBS_DIM, GLOBAL_OBS_DIM, FLAT_OBS_DIM,
40     MAX_ROBOTS, MAX_TASKS, ACTION_DIM,
41     SCENARIO_CONFIGS,
42 )
43 from environment.task_env import ImprovedTaskAllocationEnv
44 from environment.env_config import rl_config, dataset_config
45
46 logger = logging.getLogger(__name__)
47
48 # Default timestep in real time (seconds per policy step).
49 # Matches DT = TRAJECTORY_DURATION/TRAJECTORY_HORIZON = 5/64 ≈ 0.078 s but
50 # rounded up to 0.1 s to give the network inference time on embedded CPUs.
51 _DEFAULT_STEP_DT = 0.1
52
53
54 class RealWorldEnv(ImprovedTaskAllocationEnv):
55     """Wraps live sensor callbacks in the same gym.Env interface as the
56         sim.

```

```

57     The inherited parent does all obs/action-space setup and obs building.
58     This class only overrides reset() and step() to swap simulation
        physics
59     for real sensor reads and real robot commands.
60
61     Args:
62         scenario:          'warehouse' | 'search_and_rescue' |
        'maritime_security'
63         robot_state_fn:    () -> List[Dict] --- returns current robot
        state list
64         task_state_fn:     () -> List[Dict] --- returns current task
        state list
65         command_fn:        (robot_id: int, task_id: int) -> None ---
        execute assignment
66         reward_fn:         () -> float --- optional; if None the training
        loop's
67
        RewardShapingWrapper computes reward from
        self.robots/tasks
68         step_dt:           real-time seconds to wait between policy steps
69         max_steps:         episode length in steps (default from rl_config)
70         safety_layer:      optional SafetyLayer instance; applied before
        command_fn
71     """
72
73     def __init__(
74         self,
75         scenario: str,
76         robot_state_fn: Callable[[], List[Dict]],
77         task_state_fn: Callable[[], List[Dict]],
78         command_fn: Callable[[int, int], None],
79         reward_fn: Optional[Callable[[], float]] = None,
80         step_dt: float = _DEFAULT_STEP_DT,
81         max_steps: Optional[int] = None,
82         safety_layer: Optional[Any] = None,

```

```

83     ):
84         # Initialise parent: sets up observation/action spaces,
extractors,
85         # tracking dicts. A sim reset() is called internally but will be
86         # overwritten immediately by our reset().
87         super().__init__(
88             scenario=scenario,
89             config=rl_config,
90             dataset_config_param=dataset_config,
91         )
92         self._robot_state_fn = robot_state_fn
93         self._task_state_fn = task_state_fn
94         self._command_fn = command_fn
95         self._reward_fn = reward_fn
96         self._step_dt = step_dt
97         self._safety_layer = safety_layer
98         if max_steps is not None:
99             self.config.MAX_STEPS_PER_EPISODE = max_steps
100
101         # Lock for thread-safe sensor updates (ROS2 callbacks run on
other threads)
102         self._state_lock = threading.Lock()
103
104         # Latest sensor snapshot (updated by push callbacks or polled in
step)
105         self._latest_robots: List[Dict] = []
106         self._latest_tasks: List[Dict] = []
107
108         # -----
109         # Push-based sensor integration (for ROS2 / async sources)
110         # Call these from your sensor callbacks to update state in real time.
111         # step() will snapshot whatever is latest when it polls.
112         # -----
113

```

```

114 def push_robot_states(self, robot_states: List[Dict]) -> None:
115     """Thread-safe update of robot state cache (call from sensor
callbacks)."""
116     with self._state_lock:
117         self._latest_robots = robot_states
118
119 def push_task_states(self, task_states: List[Dict]) -> None:
120     """Thread-safe update of task state cache (call from sensor
callbacks)."""
121     with self._state_lock:
122         self._latest_tasks = task_states
123
124     # -----
125     # Internal helpers
126     # -----
127
128 def _poll_sensors(self) -> None:
129     """Snapshot the latest sensor state into self.robots /
self.tasks."""
130     with self._state_lock:
131         # Prefer push-cached state; fall back to pull callbacks
132         robots = self._latest_robots or self._robot_state_fn()
133         tasks = self._latest_tasks or self._task_state_fn()
134
135         self.robots = robots
136         self.tasks = tasks
137         self.num_robots = len(robots)
138         self.num_tasks = len(tasks)
139
140 def _build_tracking(self) -> None:
141     """Reset per-episode tracking structures after sensor poll."""
142     self.task_assignments = {
143         t['id']: next(
144             (r['id'] for r in self.robots

```

```

145         if r.get('current_task') == t['id']), None)
146     for t in self.tasks
147         if t.get('current_task') is not None # type: ignore[arg-type]
148     }
149     self.task_assignments = {}
150     self.robot_loads      = {r['id']: 0 for r in self.robots}
151     self.completed_tasks  = {
152         t['id'] for t in self.tasks if t.get('status') ==
TASK_STATUS_COMPLETE
153     }
154     self._rewarded_task_ids: set = set(self.completed_tasks)
155     self.robot_targets     = {r['id']: r.get('current_task') for r in
self.robots}
156     self.task_progress    = {t['id']: 0.0 for t in self.tasks}
157     self.current_step     = 0
158     self.total_energy     = 0.0
159     self.collisions       = 0
160     self.robot_collisions = 0
161     self.dynamic_obstacles = []
162     self.reallocation_count = 0
163     self.reallocation_events = []
164     self._last_realloc_step = -1
165     self._robot_travel_dist = {}
166     self._robot_waiting_time = {}
167     self._robot_prev_pos     = {}
168     self._coalition_members  = {}
169     self._coalition_wait_start = {}
170     self._abandoned_agents  = []
171     self._task_wait_times    = {}
172
173     # -----
174     # gym.Env interface
175     # -----
176

```

```

177 def reset(self, seed: Optional[int] = None, **kwargs) ->
    Tuple[np.ndarray, Dict]:
178     """Read initial sensor state and return first observation."""
179     if seed is not None:
180         np.random.seed(seed)
181
182     self._poll_sensors()
183     self._build_tracking()
184
185     logger.info(
186         f'[RealWorldEnv] reset: {self.num_robots} robots, '
187         f'{self.num_tasks} tasks (scenario={self.scenario})'
188     )
189     return self._get_observation(), self._get_info()
190
191 def step(self, action: int) -> Tuple[np.ndarray, float, bool, bool,
    Dict]:
192     """Send action to robot, wait one timestep, read new sensor state.
193
194     Args:
195         action: integer in [0, ACTION_DIM-1].
196             ACTION_DIM-1 is the no-op (keep current assignments).
197             Otherwise: robot_id = action // MAX_TASKS,
198                     task_id = action % MAX_TASKS.
199
200     Returns:
201         Standard gym 5-tuple: (obs, reward, terminated, truncated,
    info)
202     """
203     action = int(action)
204     no_op = (action >= MAX_ROBOTS * MAX_TASKS)
205
206     info: Dict[str, Any] = {'collision': False, 'task_completed':
    False}

```

```

207
208         if not no_op:
209             robot_id = action // MAX_TASKS
210             task_id = action % MAX_TASKS
211
212             # Validate bounds against current fleet/task count
213             if robot_id < self.num_robots and task_id < self.num_tasks:
214                 # Run through safety layer if one is installed
215                 if self._safety_layer is not None:
216                     safe_action, overridden =
self._safety_layer.filter_action(
217                         action, self.robots, self.tasks)
218                     if overridden:
219                         logger.debug(
220                             f'[SafetyLayer] action {action} overridden ->
{safe_action}')
221
222                         action = safe_action
223                         robot_id = action // MAX_TASKS
224                         task_id = action % MAX_TASKS
225
226                         # Issue command to physical robot
227                         self._command_fn(robot_id, task_id)
228
229                         # Update assignment tracking
230                         self.task_assignments[task_id] = robot_id
231                         self.robot_targets[robot_id] = task_id
232
233                         if task_id < len(self.tasks):
234                             self.tasks[task_id]['status'] = 'IN_PROGRESS'
235                             self.tasks[task_id]['assigned_robot'] = robot_id
236
237                         if robot_id < len(self.robots):
238                             self.robots[robot_id]['current_task'] = task_id
239
240
241             # Wait for the real world to advance one timestep
242             time.sleep(self._step_dt)
243
244
245             # Snapshot updated sensor state

```

```

239         self._poll_sensors()
240
241         # Detect new completions
242         now_complete = {
243             t['id'] for t in self.tasks if t.get('status') ==
TASK_STATUS_COMPLETE
244         }
245         new_completions = now_complete - self.completed_tasks
246         if new_completions:
247             info['task_completed'] = True
248             self.completed_tasks = now_complete
249
250         # Reward: use custom fn if provided, else return 0 so the
251         # training loop's RewardShapingWrapper computes it from
self.robots/tasks
252         reward = self._reward_fn() if self._reward_fn is not None else 0.0
253
254         self.current_step += 1
255         terminated = (len(self.completed_tasks) == self.num_tasks)
256         truncated = (self.current_step >=
self.config.MAX_STEPS_PER_EPISODE)
257
258         info.update(self._get_info())
259         return self._get_observation(), reward, terminated, truncated,
info

```

Listing B.9: Real-world environment wrapper that substitutes live sensor callbacks for the simulation physics, enabling the trained policies to run on physical robots without code changes.

```

1  """environment/analysis.py --- EDA, statistical analysis and Excel
    export."""
2  import logging
3  import warnings

```



```

4 from pathlib import Path
5 from typing import Any, Dict, List, Optional, Tuple
6
7 import matplotlib
8 matplotlib.use('Agg')
9 import matplotlib.pyplot as plt
10 from matplotlib.patches import Patch
11 import numpy as np
12 import pandas as pd
13 import seaborn as sns
14 from scipy import stats
15
16 warnings.filterwarnings('ignore')
17
18 logger = logging.getLogger(__name__)
19
20
21 # -- Helpers -----
22     -----
23
24 def _get_robot_params(scenario: str) -> Dict[str, float]:
25     """Return per-scenario robot parameters from base config."""
26     from config.base_config import SCENARIO_CONFIGS
27     sc = SCENARIO_CONFIGS[scenario]
28     return {
29         'max_speed':          sc.max_speed,
30         'battery_capacity':   sc.battery_capacity,
31         'max_payload':        sc.max_payload,
32         'observation_radius': sc.observation_radius,
33     }
34
35 def _bonferroni(p_vals: List[float], alpha: float) -> List[bool]:
36     """Bonferroni correction: return True where corrected p < alpha."""

```

```

37     n = max(len(p_vals), 1)
38     return [p < alpha / n for p in p_vals]
39
40
41 def _dep_chain_depth(task_types: List[Dict]) -> Dict[str, int]:
42     """Compute the longest dependency chain depth for each task id."""
43     id_map = {t['id']: t for t in task_types}
44     memo: Dict[str, int] = {}
45
46     def depth(tid: str, visiting: frozenset = frozenset()) -> int:
47         if tid in visiting:
48             raise ValueError(f'Circular dependency at task {tid!r}')
49         if tid in memo:
50             return memo[tid]
51         deps = id_map[tid].get('depends_on', [])
52         memo[tid] = 1 + (max(depth(d, visiting | {tid}) for d in deps) if
53             deps else 0)
54         return memo[tid]
55
56     for t in task_types:
57         try:
58             depth(t['id'])
59         except ValueError as exc:
60             logger.warning('_dep_chain_depth skipping task %r: %s',
61                 t['id'], exc)
62             memo.setdefault(t['id'], 1)
63     return memo
64
65 # -- EDA -----
66
67 def run_eda(scenarios: List[str] = None,
68             n_sample_envs: int = 5,

```

```

68         save_dir: str = './results/eda') -> Dict:
69         """
70         Exploratory Data Analysis across all three scenarios.
71
72         Produces 9 publication-ready figures:
73         1. Robot parameters comparison (bar charts with ratio annotations)
74         2. Task location heatmaps (2-D density + colour-bar + peak marker)
75         3. Task priority distribution (horizontal grouped bar)
76         4. Population distributions (per-scenario histograms)
77         5. Robot capability radar (min-max normalised per axis)
78         6. Task payload violin plot (with jittered scatter overlay)
79         7. Scenario complexity heatmap (min-max normalised)
80         8. Task dependency chain depth (motivates ordering-aware allocation)
81         9. Scenario metrics absolute values (raw grouped bar, complements Fig
82            7)
83
84         Returns
85         -----
86         all_stats : Dict
87             Nested dict keyed by scenario with collected statistics.
88         """
89         from config.base_config import SCENARIO_CONFIGS, SCENARIO_TASK_TYPES
90         from environment.task_env import ImprovedTaskAllocationEnv
91
92         scenarios = scenarios or ['warehouse', 'search_and_rescue',
93            'maritime_security']
94         Path(save_dir).mkdir(parents=True, exist_ok=True)
95
96         palette = {
97             'warehouse': '#2196F3',
98             'search_and_rescue': '#FF5722',
99             'maritime_security': '#4CAF50',
100         }
101         labels = {

```

```

100         'warehouse':          'Warehouse',
101         'search_and_rescue': 'Search & Rescue',
102         'maritime_security': 'Maritime Security',
103     }
104     grid_labels = {
105         'warehouse':          '14×14 m',
106         'search_and_rescue': '30×30 m',
107         'maritime_security': '80×80 m',
108     }
109
110     sns.set_theme(style='whitegrid', font_scale=1.0)
111
112     logger.info('=' * 80)
113     logger.info('EXPLORATORY DATA ANALYSIS')
114     logger.info('=' * 80)
115
116     # -- Data collection -----
117     -----
118     all_stats: Dict[str, Dict] = {}
119     for scenario in scenarios:
120         rp = _get_robot_params(scenario)
121         sc_stats = {
122             'robot_params':    rp,
123             'robot_counts':    [], 'task_counts':    [],
124             'speeds':          [], 'payloads':        [],
125             'batteries':        [], 'radii':          [],
126             'task_priorities': {'high': 0, 'medium': 0, 'low': 0},
127             'task_payloads':    [], 'task_locations': [],
128             'obstacle_counts': [],
129         }
130         logger.info('\n-- Scenario: %s --', labels[scenario])
131         for trial in range(n_sample_envs):
132             try:
133                 env = ImprovedTaskAllocationEnv(scenario=scenario)

```

```

133         env.reset()
134         sc_stats['robot_counts'].append(env.num_robots)
135         sc_stats['task_counts'].append(env.num_tasks)
136         sc_stats['obstacle_counts'].append(
137             len(env.environment.get('obstacles', [])))
138         for robot in env.robots:
139             sc_stats['speeds'].append(robot['max_speed'])
140             sc_stats['payloads'].append(robot['max_payload'])
141
142         sc_stats['batteries'].append(robot['battery_capacity'])
143         sc_stats['radii'].append(robot.get('radius', 0.5))
144         for task in env.tasks:
145             pri = task.get('priority', 'medium')
146             sc_stats['task_priorities'][pri] += 1
147
148         sc_stats['task_payloads'].append(task.get('payload_weight', 0))
149
150         sc_stats['task_locations'].append(task['location'].tolist())
151         except Exception as e:
152             logger.warning('EDA trial %d failed for %s: %s', trial,
153                             scenario, e)
154
155         for key in ['robot_counts', 'task_counts', 'speeds', 'batteries',
156                     'payloads']:
157             vals = sc_stats[key]
158             if vals:
159                 logger.info(' %-20s: mean=%.2f min=%.2f max=%.2f
160                             std=%.2f',
161                             key, np.mean(vals), np.min(vals),
162                             np.max(vals), np.std(vals))
163             total_tasks = sum(sc_stats['task_priorities'].values())
164             if total_tasks:
165                 for pri, cnt in sc_stats['task_priorities'].items():
166                     logger.info(' Task priority %-8s: %d/%d (%.1f%%)',

```

```

161         pri, cnt, total_tasks, 100 * cnt /
total_tasks)
162     all_stats[scenario] = sc_stats
163
164     # -- Figure 1: Robot parameters --- 2x2 bar chart with ratio
annotation -----
165     fig, axes = plt.subplots(2, 2, figsize=(14, 9))
166     fig.suptitle('Robot Parameters by Scenario', fontsize=14,
167                 fontweight='bold', y=1.01)
168     param_names = ['max_speed', 'battery_capacity', 'max_payload',
169                  'observation_radius']
170     ax_labels   = ['Max Speed (m/s)', 'Battery Capacity (Wh)',
171                  'Max Payload (kg)', 'Obs. Radius (m)']
172     x = np.arange(len(scenarios))
173     for ax, param, ax_lbl in zip(axes.flatten(), param_names, ax_labels):
174         vals    = [_get_robot_params(s)[param] for s in scenarios]
175         colors  = [palette[s] for s in scenarios]
176         bars    = ax.bar(x, vals, width=0.5, color=colors, alpha=0.88,
177                         edgecolor='white', linewidth=0.8)
178         ax.set_title(ax_lbl, fontsize=11, pad=6)
179         ax.set_ylabel(ax_lbl, fontsize=9)
180         ax.set_xticks(x)
181         ax.set_xticklabels([labels[s] for s in scenarios],
182                            rotation=15, fontsize=9, ha='right')
183         ax.spines[['top', 'right']].set_visible(False)
184         for bar, val in zip(bars, vals):
185             ax.text(bar.get_x() + bar.get_width() / 2,
186                    bar.get_height() * 1.02,
187                    f'{val:.1f}', ha='center', va='bottom', fontsize=9)
188         if len(scenarios) >= 3 and vals[0] > 0:
189             ax.annotate(f'{vals[-1]/vals[0]:.0f}x vs Warehouse',
190                       xy=(1, 0.97), xycoords='axes fraction',
191                       ha='right', va='top', fontsize=8,
192                       color='#555', style='italic')

```

```

193 plt.tight_layout()
194 fig.savefig(f'{save_dir}/robot_params_comparison.png',
195             dpi=160, bbox_inches='tight')
196 plt.close(fig)
197 logger.info('\n Saved: %s/robot_params_comparison.png', save_dir)
198
199 # -- Figure 2: Task location heatmaps with colour-bar + peak marker --
200 -----
201 fig, axes = plt.subplots(1, len(scenarios), figsize=(17, 5.5))
202 if len(scenarios) == 1:
203     axes = [axes]
204 for ax, scenario in zip(axes, scenarios):
205     locs = np.array(all_stats[scenario]['task_locations'])
206     if len(locs) >= 4:
207         h, xedges, yedges, im = ax.hist2d(
208             locs[:, 0], locs[:, 1], bins=15, cmap='YlOrRd',
209             density=True)
210         cb = plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
211         cb.set_label('Density', fontsize=8)
212         cb.ax.tick_params(labelsize=7)
213         pi = np.unravel_index(h.argmax(), h.shape)
214         peak_x = (xedges[pi[0]] + xedges[pi[0] + 1]) / 2
215         peak_y = (yedges[pi[1]] + yedges[pi[1] + 1]) / 2
216         ax.plot(peak_x, peak_y, 'b^', markersize=8, label='Peak')
217         ax.legend(fontsize=7, loc='lower right')
218         ax.set_title(f'{labels[scenario]}\nTask Location Density '
219                     f'({grid_labels[scenario]})', fontsize=10)
220         ax.set_xlabel('X position', fontsize=9)
221         ax.set_ylabel('Y position', fontsize=9)
222         ax.spines[['top', 'right']].set_visible(False)
223 plt.suptitle('Task Location Heatmaps', fontsize=13, fontweight='bold')
224 plt.tight_layout()
225 fig.savefig(f'{save_dir}/task_location_heatmaps.png',
226             dpi=160, bbox_inches='tight')

```

```

225 plt.close(fig)
226 logger.info(' Saved: %s/task_location_heatmaps.png', save_dir)
227
228 # -- Figure 3: Task priority --- horizontal grouped bar -----
229
230 fig, ax = plt.subplots(figsize=(10, 4))
231 priorities = ['high', 'medium', 'low']
232 pri_colors = {'high': '#E53935', 'medium': '#FB8C00', 'low':
233               '#43A047'}
234 bar_h = 0.22
235 y_base = np.arange(len(scenarios))
236 for pi, pri in enumerate(priorities):
237     offsets = y_base + (pi - 1) * bar_h
238     vals = []
239     for sc in scenarios:
240         total = sum(all_stats[sc]['task_priorities'].values())
241         cnt = all_stats[sc]['task_priorities'].get(pri, 0)
242         vals.append(100 * cnt / total if total > 0 else 0)
243     bars = ax.barh(offsets, vals, height=bar_h, color=pri_colors[pri],
244                   alpha=0.88, label=pri.capitalize(),
245                   edgecolor='white')
246     for bar, val in zip(bars, vals):
247         if val > 1:
248             ax.text(val + 0.5, bar.get_y() + bar.get_height() / 2,
249                   f'{val:.1f}%', va='center', fontsize=8)
250 ax.set_yticks(y_base)
251 ax.set_yticklabels([labels[s] for s in scenarios], fontsize=10)
252 ax.set_xlabel('Percentage of tasks (%)', fontsize=10)
253 ax.set_title('Task Priority Distribution by Scenario',
254             fontsize=12, fontweight='bold')
255 ax.legend(loc='lower right', fontsize=9)
256 ax.spines[['top', 'right']].set_visible(False)
257 ax.set_xlim(0, 65)
258 ax.axvline(33.3, color='gray', lw=0.7, ls='--', alpha=0.5)

```



```

256 plt.tight_layout()
257 fig.savefig(f'{save_dir}/task_priority_distribution.png',
258             dpi=160, bbox_inches='tight')
259 plt.close(fig)
260 logger.info(' Saved: %s/task_priority_distribution.png', save_dir)
261
262 # -- Figure 4: Population distributions --- per-scenario grid -----
263
264 fig, axes = plt.subplots(len(scenarios), 2,
265                           figsize=(12, 4 * len(scenarios)))
266 fig.suptitle('Population Distributions by Scenario',
267              fontsize=13, fontweight='bold')
268 if len(scenarios) == 1:
269     axes = [axes]
270 for row, scenario in enumerate(scenarios):
271     for col, (key, xlabel, bins) in enumerate([
272         ('robot_counts', 'Robot Count', 8),
273         ('task_counts', 'Task Count', 10),
274     ]):
275         ax = axes[row][col]
276         vals = all_stats[scenario][key]
277         if vals:
278             ax.hist(vals, bins=bins, color=palette[scenario],
279                     alpha=0.82, edgecolor='white', lw=0.6)
280             ax.axvline(np.mean(vals), color='black', lw=1.2, ls='--',
281                        label=f'Mean: {np.mean(vals):.1f}')
282             ax.axvline(np.median(vals), color='#555', lw=1.0, ls=':',
283                        label=f'Median: {np.median(vals):.1f}')
284             ax.legend(fontsize=8)
285             summary = (f"n={len(vals)} std={np.std(vals):.1f}\n"
286                        f"min={np.min(vals):.0f}"
287                        f"max={np.max(vals):.0f}")
288             ax.text(0.97, 0.97, summary, transform=ax.transAxes,
289                     fontsize=7, va='top', ha='right',

```

```

288         bbox=dict(boxstyle='round,pad=0.3',
289                   fc='white', ec='#ccc', alpha=0.8))
290         ax.set_title(f'{labels[scenario]} --- {xlabel}', fontsize=10)
291         ax.set_xlabel(xlabel, fontsize=9)
292         ax.set_ylabel('Frequency', fontsize=9)
293         ax.spines[['top', 'right']].set_visible(False)
294     plt.tight_layout()
295     fig.savefig(f'{save_dir}/population_distributions.png',
296               dpi=160, bbox_inches='tight')
297     plt.close(fig)
298     logger.info(' Saved: %s/population_distributions.png', save_dir)
299
300     # -- Figure 5: Robot capability radar -----
301     -----
302     radar_params = ['max_speed', 'battery_capacity', 'max_payload',
303                   'observation_radius']
304     radar_labels = ['Speed', 'Battery', 'Payload', 'Obs Radius']
305     # Per-axis min-max normalisation so all scenarios fill the chart
306     # fairly
307     _all_rp = np.array([[_get_robot_params(sc)[p] for p in radar_params]
308                       for sc in scenarios])
309     _ax_min   = _all_rp.min(axis=0)
310     _ax_max   = _all_rp.max(axis=0)
311     _ax_range = np.where(_ax_max - _ax_min < 1e-9, 1.0, _ax_max - _ax_min)
312     N         = len(radar_params)
313     angles = np.linspace(0, 2 * np.pi, N, endpoint=False).tolist()
314     angles += angles[:1]
315     fig, ax = plt.subplots(figsize=(7, 7), subplot_kw=dict(polar=True))
316     ax.set_theta_offset(np.pi / 2)
317     ax.set_theta_direction(-1)
318     ax.set_xticks(angles[:-1])
319     ax.set_xticklabels(radar_labels, fontsize=10)
320     ax.set_rlabel_position(30)
321     ax.set_ylim(0, 1)

```

```

320     ax.yaxis.set_tick_params(labelsize=7)
321     ax.spines['polar'].set_visible(False)
322     for scenario in scenarios:
323         rp = _get_robot_params(scenario)
324         vals = [(rp[p] - _ax_min[i]) / _ax_range[i]
325                 for i, p in enumerate(radar_params)]
326         vals += vals[:1]
327         ax.plot(angles, vals, lw=2, color=palette[scenario],
328                 label=labels[scenario])
329         ax.fill(angles, vals, alpha=0.12, color=palette[scenario])
330     ax.legend(loc='upper right', bbox_to_anchor=(1.3, 1.1), fontsize=10)
331     ax.set_title('Robot Capability Radar\n(min-max normalised per axis)',
332                 fontsize=12, fontweight='bold', pad=20)
333     plt.tight_layout()
334     fig.savefig(f'{save_dir}/robot_capability_radar.png',
335                 dpi=160, bbox_inches='tight')
336     plt.close(fig)
337     logger.info(' Saved: %s/robot_capability_radar.png', save_dir)
338
339     # -- Figure 6: Task payload violin with jittered scatter overlay -----
340     -----
341     fig, ax = plt.subplots(figsize=(10, 5))
342     payload_data = [all_stats[sc][i]['task_payloads'] for sc in scenarios]
343     non_empty = [(i + 1, d, sc)
344                  for i, (d, sc) in enumerate(zip(payload_data,
345                                                    scenarios))]
346
347     if d]
348
349     if non_empty:
350         parts = ax.violinplot(
351             [x[1] for x in non_empty],
352             positions=[x[0] for x in non_empty],
353             widths=0.55, showmeans=True, showmedians=True,
354             showextrema=True)
355         for pc, (_, _, sc) in zip(parts['bodies'], non_empty):

```

```

351         pc.set_facecolor(palette[sc])
352         pc.set_alpha(0.7)
353         parts['cmeans'].set_color('black')
354         parts['cmedians'].set_color('#333')
355         _jitter_rng = np.random.default_rng(seed=42)
356         for pos, data, sc in non_empty:
357             jitter = _jitter_rng.normal(0, 0.04, size=len(data))
358             ax.scatter(np.full(len(data), pos) + jitter, data,
359                       alpha=0.25, s=8, color=palette[sc], zorder=2)
360         ax.set_xticks(range(1, len(scenarios) + 1))
361         ax.set_xticklabels([labels[s] for s in scenarios], fontsize=10)
362         ax.set_ylabel('Task payload weight (kg)', fontsize=10)
363         ax.set_title('Task Payload Distribution by Scenario',
364                     fontsize=12, fontweight='bold')
365         ax.spines[['top', 'right']].set_visible(False)
366         plt.tight_layout()
367         fig.savefig(f'{save_dir}/task_payload_violin.png',
368                   dpi=160, bbox_inches='tight')
369         plt.close(fig)
370         logger.info(' Saved: %s/task_payload_violin.png', save_dir)
371
372         # -- Figure 7: Scenario complexity heatmap (min-max normalised) -----
373         -----
374         metrics_matrix = []
375         for sc in scenarios:
376             st = all_stats[sc]
377             rp = _get_robot_params(sc)
378             tt = sum(st['task_priorities'].values())
379             hp = 100 * st['task_priorities']['high'] / tt if tt else 0
380             metrics_matrix.append([
381                 np.mean(st['robot_counts']) if st['robot_counts'] else
382                 0,
383                 np.mean(st['task_counts']) if st['task_counts'] else
384                 0,

```

```

382         np.mean(st['obstacle_counts']) if st['obstacle_counts'] else
0,
383         hp,
384         rp['max_speed'],
385         rp['battery_capacity'] / 100,
386         rp['max_payload'] / 100,
387     ])
388     col_labels = ['Avg Robots', 'Avg Tasks', 'Avg Obstacles',
389                  'High Priority %', 'Max Speed (m/s)',
390                  'Battery (×100 Wh)', 'Payload (×100 kg)']
391     mat = np.array(metrics_matrix, dtype=float)
392     mat_range = mat.max(0) - mat.min(0)
393     mat_norm = (mat - mat.min(0)) / (mat_range + 1e-9)
394     fig, ax = plt.subplots(figsize=(13, 3.5))
395     ax.imshow(mat_norm, cmap='Blues', aspect='auto', vmin=0, vmax=1)
396     ax.set_xticks(np.arange(len(col_labels)))
397     ax.set_xticklabels(col_labels, rotation=30, ha='right', fontsize=9)
398     ax.set_yticks(np.arange(len(scenarios)))
399     ax.set_yticklabels([labels[s] for s in scenarios], fontsize=10)
400     ax.set_title('Scenario Complexity Overview (min-max normalised)',
401                 fontsize=11, fontweight='bold', pad=10)
402     for i in range(len(scenarios)):
403         for j in range(len(col_labels)):
404             raw = mat[i, j]
405             txt = f'{raw:.1f}' if raw < 100 else f'{raw:.0f}'
406             tc = 'white' if mat_norm[i, j] > 0.6 else 'black'
407             ax.text(j, i, txt, ha='center', va='center', fontsize=9,
408                    color=tc)
409     plt.tight_layout()
410     fig.savefig(f'{save_dir}/scenario_complexity_heatmap.png',
411                dpi=160, bbox_inches='tight')
412     plt.close(fig)
413     logger.info(' Saved: %s/scenario_complexity_heatmap.png', save_dir)

```

```

414 # -- Figure 8: Task dependency chain depth (NEW) -----
415 # Shows how many tasks have ordering constraints and the longest
    dependency
416 # chain per scenario --- directly motivates ordering-aware allocation.
417 fig, axes = plt.subplots(1, len(scenarios),
418                          figsize=(6 * len(scenarios), 5),
    sharey=False)
419 if len(scenarios) == 1:
420     axes = [axes]
421 fig.suptitle('Task Dependency Structure by Scenario\n'
422             '(depth = longest prerequisite chain)',
423             fontsize=13, fontweight='bold')
424
425 for ax, scenario in zip(axes, scenarios):
426     task_types = SCENARIO_TASK_TYPES.get(scenario, [])
427     if not task_types:
428         ax.set_title(labels[scenario])
429         ax.text(0.5, 0.5, 'No task types', transform=ax.transAxes,
430                ha='center', va='center')
431         continue
432
433     depths = _dep_chain_depth(task_types)
434     task_ids = [t['id'] for t in task_types]
435     task_names = [t['name'] for t in task_types]
436     dep_counts = [len(t.get('depends_on', [])) for t in task_types]
437     depth_vals = [depths[tid] for tid in task_ids]
438
439     # Color by depth: depth 1 = independent, deeper = more constrained
440     max_depth = max(depth_vals)
441     cmap =
    matplotlib.colormaps.get_cmap('YlOrRd').resampled(max(max_depth, 1))
442     bar_colors = [cmap((d - 1) / max(max_depth - 1, 1)) for d in
    depth_vals]

```

```

443
444     y_pos = np.arange(len(task_ids))
445     bars = ax.barh(y_pos, depth_vals, color=bar_colors,
446                   edgecolor='white', linewidth=0.7, alpha=0.9)
447
448     # Annotate with dependency count
449     for bar, dc_val, dv in zip(bars, dep_counts, depth_vals):
450         label = f'depth {dv}' + (f' ({dc_val} dep)' if dc_val > 0
451     else ' (free)')
452         ax.text(dv + 0.05, bar.get_y() + bar.get_height() / 2,
453               label, va='center', fontsize=7)
454
455     ax.set_yticks(y_pos)
456     ax.set_yticklabels(
457         [f'{tid}: {name[:22]}' for tid, name in zip(task_ids,
458             task_names)],
459         fontsize=8)
460     ax.set_xlabel('Dependency Chain Depth', fontsize=9)
461     ax.set_title(
462         f'{labels[scenario]}\n'
463         f'{sum(1 for d in depth_vals if d > 1)} constrained / '
464         f'{len(task_ids)} tasks | max depth {max_depth}',
465         fontsize=10)
466     ax.set_xlim(0, max_depth + 1.8)
467     ax.spines[['top', 'right']].set_visible(False)
468     ax.invert_yaxis()
469
470     # Colour legend
471     legend_elements = [
472         Patch(facecolor=cmap(0.0), label='Depth 1 (independent)'),
473         Patch(facecolor=cmap(0.99), label=f'Depth {max_depth} (most
474     constrained)'),
475     ]
476     ax.legend(handles=legend_elements, fontsize=7, loc='lower right')

```

```

474
475 plt.tight_layout()
476 fig.savefig(f'{save_dir}/task_dependency_chains.png',
477             dpi=160, bbox_inches='tight')
478 plt.close(fig)
479 logger.info(' Saved: %s/task_dependency_chains.png', save_dir)
480
481 # -- Figure 9: Scenario metrics --- absolute-value grouped bar chart -
482 -----
483 # Companion to Figure 7's normalised heatmap; shows raw magnitudes so
484 # readers can compare actual numbers without back-calculating from
485 colours.
486
487 fig, ax = plt.subplots(figsize=(15, 5))
488 x      = np.arange(len(col_labels))
489 width = 0.25
490 sc_colors = [palette[sc] for sc in scenarios]
491 for i, (sc, color) in enumerate(zip(scenarios, sc_colors)):
492     offset = x + (i - 1) * width
493     bars = ax.bar(offset, mat[i], width=width, color=color,
494                   alpha=0.85, label=labels[sc], edgecolor='white')
495     for bar, val in zip(bars, mat[i]):
496         ax.text(bar.get_x() + bar.get_width() / 2,
497                 bar.get_height() * 1.01,
498                 f'{val:.1f}' if val < 100 else f'{val:.0f}',
499                 ha='center', va='bottom', fontsize=7, rotation=55)
500
501 ax.set_xticks(x)
502 ax.set_xticklabels(col_labels, rotation=25, ha='right', fontsize=9)
503 ax.set_title('Scenario Metrics --- Absolute Values',
504             fontsize=12, fontweight='bold')
505 ax.set_ylabel('Value (units vary per metric)', fontsize=9)
506 ax.legend(fontsize=9)
507 ax.spines[['top', 'right']].set_visible(False)
508 plt.tight_layout()
509 fig.savefig(f'{save_dir}/scenario_metrics_absolute.png',

```



```

506         dpi=160, bbox_inches='tight')
507     plt.close(fig)
508     logger.info('  Saved: %s/scenario_metrics_absolute.png', save_dir)
509
510     logger.info('\nEDA complete. All 9 plots saved to: %s/', save_dir)
511     logger.info('=' * 80)
512     return all_stats
513
514
515 # -- Statistical analysis -----
516     -----
517 def run_statistical_analysis(all_results: Dict, alpha: float = 0.05) ->
518     Dict:
519     """
520
521     Run comprehensive statistical analysis across all scenarios and
522     algorithms.
523
524     Returns a nested dict:
525     stats[scenario]['rl_pairwise']      -- pairwise RL reward comparisons
526     stats[scenario]['rl_vs_baseline']   -- each RL algo vs each baseline
527     stats[scenario]['diff_pairwise']    -- pairwise diffusion loss
528     comparisons
529     stats[scenario]['rl_descriptive']   -- per-algo descriptive stats
530     stats[scenario]['diff_descriptive'] -- per-model descriptive stats
531     """
532     from training.statistical_eval import StatisticalEvaluator
533
534     output: Dict = {}
535     for scenario, sc_data in all_results.items():
536         rl_models = sc_data.get('rl_models', {})
537         diff_models = sc_data.get('diffusion_models', {})
538         sc_stats: Dict = {}

```

```

536     # -- RL descriptive statistics -----
537     --
538     rl_desc: Dict = {}
539     for algo, d in rl_models.items():
540         rews = np.array(d.get('rewards', [0]), dtype=float)
541         ci    = StatisticalEvaluator.bootstrap_ci(rews,
n_bootstrap=2000) \
542             if len(rews) >= 2 else (float(rews[0]) if len(rews)
else 0.0,) * 2
543         if len(rews) >= 3:
544             sw_stat, sw_pval = stats.shapiro(rews)
545             shapiro_stat = float(sw_stat)
546             shapiro_pval = float(sw_pval)
547             is_normal     = bool(sw_pval >= alpha)
548         else:
549             shapiro_stat = float('nan')
550             shapiro_pval = float('nan')
551             is_normal     = None
552         rl_desc[algo] = {
553             'n':          len(rews),
554             'mean':       float(np.mean(rews)),
555             'median':     float(np.median(rews)),
556             'std':        float(np.std(rews, ddof=1)) if len(rews)
> 1 else 0.0,
557             'sem':       float(np.std(rews, ddof=1) /
np.sqrt(len(rews)))
558             if len(rews) > 1 else 0.0,
559             'min':       float(np.min(rews)),
560             'max':       float(np.max(rews)),
561             'q25':       float(np.percentile(rews, 25)),
562             'q75':       float(np.percentile(rews, 75)),
563             'ci_low':    float(ci[0]),
564             'ci_high':   float(ci[1]),
565             'cv':        float(np.std(rews, ddof=1) /

```

```

( abs(np.mean(rews)) + 1e-9)),
565         'shapiro_stat': shapiro_stat,
566         'shapiro_pval': shapiro_pval,
567         'is_normal': is_normal,
568     }
569     sc_stats['rl_descriptive'] = rl_desc
570
571     # -- RL pairwise tests -----
--
572     algos = [a for a, d in rl_models.items()
573              if len(d.get('rewards', [])) >= 5]
574     pairs: List[Dict] = []
575     p_vals: List[float] = []
576     for i in range(len(algos)):
577         for j in range(i + 1, len(algos)):
578             a1, a2 = algos[i], algos[j]
579             s1 = np.array(rl_models[a1]['rewards'], dtype=float)
580             s2 = np.array(rl_models[a2]['rewards'], dtype=float)
581             try:
582                 t_res = StatisticalEvaluator.welch_t_test(s1, s2,
alpha)
583                 u_res = StatisticalEvaluator.mann_whitney_u(s1, s2,
alpha)
584                 d_eff = StatisticalEvaluator.cohens_d(s1, s2)
585                 ci1 = StatisticalEvaluator.bootstrap_ci(s1,
n_bootstrap=2000)
586                 ci2 = StatisticalEvaluator.bootstrap_ci(s2,
n_bootstrap=2000)
587                 pairs.append({
588                     'algo1': a1,
589                     'algo2': a2,
590                     'mean1': float(np.mean(s1)),
591                     'mean2': float(np.mean(s2)),
592                     'std1': float(np.std(s1, ddof=1)) if

```

```

len(s1) > 1 else 0.0,
593         'std2':          float(np.std(s2, ddof=1)) if
len(s2) > 1 else 0.0,
594         'ci1_low':       ci1[0], 'ci1_high': ci1[1],
595         'ci2_low':       ci2[0], 'ci2_high': ci2[1],
596         't_stat':        t_res['t_stat'],
597         't_pval':        t_res['p_value'],
598         'u_stat':        u_res['u_stat'],
599         'u_pval':        u_res['p_value'],
600         'cohens_d':      d_eff,
601         'effect_label':  ('large' if abs(d_eff) >= 0.8
else
602                             'medium' if abs(d_eff) >= 0.5
else
603                             'small' if abs(d_eff) >= 0.2
else
604                             'negligible'),
605         't_sig_raw':     t_res['p_value'] < alpha,
606         'u_sig_raw':     u_res['p_value'] < alpha,
607     })
608     p_vals.append(t_res['p_value'])
609     except Exception as e:
610         logger.warning('Pairwise stat failed %s vs %s: %s',
a1, a2, e)
611
612     if p_vals:
613         bonf = _bonferroni(p_vals, alpha)
614         for k, pair in enumerate(pairs):
615             pair['bonferroni_sig'] = bonf[k]
616     sc_stats['rl_pairwise'] = pairs
617
618     # -- RL vs baseline -----
619     --
620     bc = sc_data.get('baseline_comparison', {})

```

```

620     rl_vs_base: List[Dict] = []
621     for algo, d in rl_models.items():
622         comps = np.array(d.get('completions', [0.0]), dtype=float)
623         rews = np.array(d.get('rewards', [0.0]), dtype=float)
624         rl_completion = float(np.mean(comps)) if len(comps) > 0 else
float('nan')
625         entry: Dict = {
626             'algo': algo,
627             'rl_mean_reward': float(np.mean(rews)),
628             'rl_mean_completion': rl_completion,
629         }
630         for base in ['hungarian', 'greedy', 'cbba', 'random']:
631             base_info = bc.get(base, {})
632             base_completion = base_info.get('efficiency',
float('nan'))
633             base_cost = base_info.get('total_cost',
float('nan'))
634             entry[f'{base}_completion'] = base_completion
635             entry[f'{base}_cost'] = base_cost
636             if (not np.isnan(rl_completion) and not
np.isnan(base_completion)
637                 and base_completion > 0):
638                 entry[f'{base}_pct_improvement'] = (
639                     (rl_completion - base_completion) /
base_completion * 100)
640             else:
641                 entry[f'{base}_pct_improvement'] = float('nan')
642             rl_vs_base.append(entry)
643             sc_stats['rl_vs_baseline'] = rl_vs_base
644
645             # -- Diffusion descriptive + pairwise -----
-
646             diff_desc: Dict = {}
647             for key, d in diff_models.items():

```

```

648         losses = np.array(d.get('losses', [float('nan')]),
dtype=float)
649         losses = losses[~np.isnan(losses)]
650         if len(losses) == 0:
651             losses = np.array([float('nan')])
652         final10 = losses[-10:] if len(losses) >= 10 else losses
653         threshold = float(np.mean(final10)) * 1.10 if len(final10)
else float('nan')
654         converge_ep = (next((i for i, lv in enumerate(losses)
655                             if lv <= threshold), len(losses))
656                        if not np.isnan(threshold) else float('nan'))
657         diff_desc[key] = {
658             'final_loss': float(np.mean(final10)) if
len(final10) else float('nan'),
659             'loss_std': float(np.std(final10, ddof=1)) if
len(final10) > 1 else 0.0,
660             'min_loss': float(np.min(losses)) if len(losses)
else float('nan'),
661             'max_loss': float(np.max(losses)) if len(losses)
else float('nan'),
662             'loss_reduction': (float(losses[0] - np.mean(final10))
663                               / (abs(losses[0]) + 1e-9) * 100
664                               if len(losses) > 1 else
float('nan')),
665             'epochs': len(losses),
666             'convergence_ep': (int(converge_ep)
667                               if not np.isnan(converge_ep) else
'N/A'),
668             'num_trajectories': d.get('num_trajectories', 'N/A'),
669         }
670         sc_stats['diff_descriptive'] = diff_desc
671
672         diff_algos_avail = [k for k, d in diff_models.items()
673                             if not np.isnan(d.get('final_loss',

```

```

float('nan')))]
674     diff_pairs: List[Dict] = []
675     dp_vals: List[float] = []
676     for i in range(len(diff_algos_avail)):
677         for j in range(i + 1, len(diff_algos_avail)):
678             k1, k2 = diff_algos_avail[i], diff_algos_avail[j]
679             l1 = np.array(diff_models[k1].get('losses', []),
dtype=float)
680             l2 = np.array(diff_models[k2].get('losses', []),
dtype=float)
681             l1 = l1[~np.isnan(l1)]; l2 = l2[~np.isnan(l2)]
682             if len(l1) < 3 or len(l2) < 3:
683                 continue
684             try:
685                 t_res = StatisticalEvaluator.welch_t_test(l1, l2,
alpha)
686                 d_eff = StatisticalEvaluator.cohens_d(l1, l2)
687                 diff_pairs.append({
688                     'model1':      k1,   'model2': k2,
689                     'mean1':      float(np.mean(l1)),
690                     'mean2':      float(np.mean(l2)),
691                     'std1':      float(np.std(l1, ddof=1)) if
len(l1) > 1 else 0.0,
692                     'std2':      float(np.std(l2, ddof=1)) if
len(l2) > 1 else 0.0,
693                     't_stat':      t_res['t_stat'],
694                     't_pval':      t_res['p_value'],
695                     'cohens_d':    d_eff,
696                     'effect_label': ('large' if abs(d_eff) >= 0.8
else
697                                     'medium' if abs(d_eff) >= 0.5
else
698                                     'small' if abs(d_eff) >= 0.2
else

```

```

699                                     'negligible'),
700                                     't_sig_raw': t_res['p_value'] < alpha,
701                                     })
702                                     dp_vals.append(t_res['p_value'])
703                                     except Exception:
704                                         pass
705                                     if dp_vals:
706                                         db_bonf = _bonferroni(dp_vals, alpha)
707                                         for k, dp in enumerate(diff_pairs):
708                                             dp['bonferroni_sig'] = db_bonf[k]
709                                     sc_stats['diff_pairwise'] = diff_pairs
710                                     output[scenario] = sc_stats
711
712     return output
713
714
715 # -- Excel export -----
716 -----
717 def export_results_to_excel(all_results: Dict,
718                             filepath: str =
719                             './results/training_results.xlsx',
720                             statistical_results: Dict = None) -> str:
721     """
722     Export training results to a publication-quality Excel workbook.
723
724     Sheets (10 total):
725     1. Summary -- headline metrics per scenario
726     2. RL Results -- full descriptive statistics per algorithm
727     3. Diffusion Results -- loss curves + convergence per model
728     4. Baselines -- all 4 baselines per scenario
729     5. Stat Tests (RL) -- Welch t, Mann-Whitney U, Cohen's d,
730                                     bootstrap CI,
731                                     Bonferroni (pairwise)

```



```

730     6. Stat Tests (Diff)      -- same for diffusion loss distributions
731     7. RL vs Baselines      -- completion rate comparison
732     8. Effect Sizes (RL)    -- Cohen's d matrix (all pairs, all
        scenarios)
733     9. Convergence Analysis -- loss reduction %, convergence episode
734     10. Energy & Reallocation
735     """
736     from config.base_config import RL_ALGOS, DIFF_ALGOS, MAX_TASKS,
        MAX_STEPS
737
738     try:
739         from openpyxl import Workbook
740         from openpyxl.styles import Font, PatternFill, Alignment, Border,
        Side
741         from openpyxl.utils import get_column_letter
742     except ImportError:
743         logger.error('openpyxl not installed --- run: pip install
        openpyxl')
744         return ''
745
746     if statistical_results is None:
747         logger.info(' Running statistical analysis...')
748         statistical_results = run_statistical_analysis(all_results)
749
750     Path(filepath).parent.mkdir(parents=True, exist_ok=True)
751     wb = Workbook()
752
753     # -- Style constants -----
754     --
755     _HEADER      = PatternFill('solid', start_color='0D47A1')
756     _SUBHEADER   = PatternFill('solid', start_color='1565C0')
757     _ALT         = PatternFill('solid', start_color='E3F2FD')
758     _BEST        = PatternFill('solid', start_color='C8E6C9')
759     _SIG         = PatternFill('solid', start_color='FFF9C4')

```

```

759     _WARN      = PatternFill('solid', start_color='FFCDD2')
760     _LARGE     = PatternFill('solid', start_color='E1BEE7')
761
762     _HFONT = Font(name='Calibri', bold=True, color='FFFFFF', size=10)
763     _BFONT = Font(name='Calibri', bold=True, size=10)
764     _DFONT = Font(name='Calibri', size=10)
765     _CTR    = Alignment(horizontal='center', vertical='center',
766                          wrap_text=True)
767     _LFT    = Alignment(horizontal='left', vertical='center',
768                          wrap_text=True)
769
770     _ts      = Side(style='thin')
771     _BDR     = Border(left=_ts, right=_ts, top=_ts, bottom=_ts)
772
773     FMT_2DP  = '#,##0.00'
774     FMT_4DP  = '0.0000'
775     FMT_6DP  = '0.000000'
776     FMT_PCT  = '0.00%'
777     FMT_SCI  = '0.00E+00'
778
779     def hdr(ws, row, col, val, w=16, sub=False):
780         c = ws.cell(row=row, column=col, value=val)
781         c.font      = _HFONT
782         c.fill      = _SUBHEADER if sub else _HEADER
783         c.alignment = _CTR
784         c.border     = _BDR
785         ws.column_dimensions[get_column_letter(col)].width = w
786         ws.row_dimensions[row].height = 28
787
788     def dc(ws, row, col, val, fill=None, fmt=None, bold=False):
789         if isinstance(val, float) and np.isnan(val):
790             val = '---'
791         c = ws.cell(row=row, column=col, value=val)
792         c.font      = _BFONT if bold else _DFONT
793         c.alignment = _CTR

```

```

791         c.border      = _BDR
792         if fill: c.fill = fill
793         if fmt:  c.number_format = fmt
794         return c
795
796     def sig_fill(pval, bonf_sig=None):
797         if bonf_sig is not None:
798             return _BEST if bonf_sig else _WARN
799         if pval is None or (isinstance(pval, float) and np.isnan(pval)):
800             return None
801         return _SIG if pval < 0.05 else None
802
803     def eff_fill(label):
804         return _LARGE if label == 'large' else (_BEST if label ==
805         'medium' else None)
806
807     def nan_r(v, dp=4):
808         if v is None: return '---'
809         if isinstance(v, float) and np.isnan(v): return '---'
810         return round(float(v), dp)
811
812     scenarios = list(all_results.keys())
813     base_names = ['random', 'greedy', 'hungarian', 'cbba']
814
815     # -- Sheet 1: Executive Summary -----
816     --
817     ws1 = wb.active; ws1.title = 'Summary'
818     cols1 = ['Scenario', 'RL Models', 'Diff Models',
819             'Best RL Algo', 'Best RL Mean Reward', 'Best RL 95% CI',
820             'Best Completion %', 'Best RL Std Dev',
821             'Best Diff Model', 'Best Diff Loss',
822             'Hungarian Cost', 'CBBA Cost', 'Greedy Cost']
823     widths1 = [22, 12, 14, 14, 20, 22, 18, 16, 24, 16, 16, 14, 14]
824     for c, (h, w) in enumerate(zip(cols1, widths1), 1):

```

```

823         hdr(ws1, 1, c, h, w)
824     ws1.freeze_panes = 'A2'
825     ws1.auto_filter.ref = ws1.dimensions
826
827     for r, scenario in enumerate(scenarios, 2):
828         sc      = all_results[scenario]
829         rl_m     = sc.get('rl_models', {})
830         di_m     = sc.get('diffusion_models', {})
831         st       = statistical_results.get(scenario, {})
832         rl_d     = st.get('rl_descriptive', {})
833         bc       = sc.get('baseline_comparison', {})
834         fill     = _ALT if r % 2 == 0 else None
835         best_algo = max(rl_m, key=lambda k: rl_m[k]['avg_reward'],
836         default='---')
837         bd       = rl_d.get(best_algo, {})
838         best_diff = min(di_m, key=lambda k: di_m[k].get('final_loss',
839         1e9), default='---')
840         dc(ws1, r, 1, scenario.replace('_', ' ').title(), fill,
841         bold=True)
842         dc(ws1, r, 2, len(rl_m), fill)
843         dc(ws1, r, 3, len(di_m), fill)
844         dc(ws1, r, 4, best_algo, fill, bold=True)
845         dc(ws1, r, 5, nan_r(bd.get('mean'), 3), fill, FMT_2DP)
846         dc(ws1, r, 6, f"[{nan_r(bd.get('ci_low'),3)},
847         {nan_r(bd.get('ci_high'),3)}]", fill)
848         dc(ws1, r, 7, nan_r(rl_m.get(best_algo,
849         {}).get('avg_completion', 0), 4),
850         fill, FMT_PCT)
851         dc(ws1, r, 8, nan_r(bd.get('std'), 3), fill, FMT_2DP)
852         dc(ws1, r, 9, best_diff, fill)
853         dc(ws1, r, 10, nan_r(di_m.get(best_diff, {}).get('final_loss'),
854         6), fill, FMT_6DP)
855         for ci, bn in enumerate(['hungarian', 'cbba', 'greedy'], 11):
856             _bc_val = bc.get(bn, {}).get('avg_cost',

```

```

851         bc.get(bn, {}).get('total_cost'))
852         dc(ws1, r, ci, nan_r(_bc_val, 2), fill, FMT_2DP)
853
854     # -- Sheet 2: RL Results -----
855     --
856     ws2 = wb.create_sheet('RL Results')
857     cols2 = ['Scenario', 'Algorithm', 'N Episodes',
858             'Mean Reward', 'Median', 'Std Dev', 'SEM',
859             'Min', 'Max', 'Q25', 'Q75', 'IQR',
860             '95% CI Low', '95% CI High', 'CV (%)',
861             'Avg Completion %', 'Avg Collisions', 'Best Episode',
862             'Avg Energy (Wh)', 'Energy Efficiency (tasks/Wh)',
863             'Avg Reallocations/Ep']
864     widths2 = [20, 12, 12, 14, 12, 12, 10, 12, 12, 12, 10, 14, 14, 10,
865               18, 16, 14, 18, 26, 22]
866     for c, (h, w) in enumerate(zip(cols2, widths2), 1):
867         hdr(ws2, 1, c, h, w)
868     ws2.freeze_panes = 'A2'
869     ws2.auto_filter.ref = ws2.dimensions
870
871     row = 2
872     for scenario in scenarios:
873         rl_m = all_results[scenario].get('rl_models', {})
874         rl_d = statistical_results.get(scenario,
875                                       {}).get('rl_descriptive', {})
876         best = max(rl_m, key=lambda k: rl_m[k]['avg_reward'],
877                   default=None)
878         for algo in RL_ALGOS:
879             if algo not in rl_m: continue
880             d = rl_m[algo]
881             desc = rl_d.get(algo, {})
882             fill = _BEST if algo == best else (_ALT if row % 2 == 0 else
883                                               None)
884             rews = d.get('rewards', [0])

```

```

881         dc(ws2, row, 1, scenario.replace('_', ' ').title(), fill)
882         dc(ws2, row, 2, algo, fill, bold=(algo == best))
883         dc(ws2, row, 3, desc.get('n', len(rews)), fill)
884         dc(ws2, row, 4, nan_r(desc.get('mean'), 4), fill, FMT_4DP)
885         dc(ws2, row, 5, nan_r(desc.get('median'), 4), fill, FMT_4DP)
886         dc(ws2, row, 6, nan_r(desc.get('std'), 4), fill, FMT_4DP)
887         dc(ws2, row, 7, nan_r(desc.get('sem'), 4), fill, FMT_4DP)
888         dc(ws2, row, 8, nan_r(desc.get('min'), 3), fill, FMT_2DP)
889         dc(ws2, row, 9, nan_r(desc.get('max'), 3), fill, FMT_2DP)
890         dc(ws2, row, 10, nan_r(desc.get('q25'), 3), fill, FMT_2DP)
891         dc(ws2, row, 11, nan_r(desc.get('q75'), 3), fill, FMT_2DP)
892         iqr = (desc.get('q75', 0) or 0) - (desc.get('q25', 0) or 0)
893         dc(ws2, row, 12, nan_r(iqr, 3), fill,
FMT_2DP)
894         dc(ws2, row, 13, nan_r(desc.get('ci_low'), 4), fill,
FMT_4DP)
895         dc(ws2, row, 14, nan_r(desc.get('ci_high'), 4), fill,
FMT_4DP)
896         cv = (desc.get('cv') or 0) * 100
897         dc(ws2, row, 15, nan_r(cv, 2), fill,
FMT_2DP)
898         dc(ws2, row, 16, nan_r(d.get('avg_completion', 0), 4), fill,
FMT_PCT)
899         cols_v = d.get('collisions', [0])
900         dc(ws2, row, 17, nan_r(float(np.mean(cols_v)), 2), fill,
FMT_2DP)
901         dc(ws2, row, 18, nan_r(float(max(rews)) if rews else 0, 3),
fill, FMT_4DP)
902         dc(ws2, row, 19, nan_r(d.get('avg_energy', float('nan')), 4),
fill, FMT_4DP)
903         dc(ws2, row, 20, nan_r(d.get('avg_energy_efficiency',
float('nan')), 6), fill, FMT_6DP)
904         dc(ws2, row, 21, nan_r(d.get('avg_reallocation_rate', 0.0),
2), fill, FMT_2DP)

```

```

905         row += 1
906
907     # -- Sheet 3: Diffusion Results -----
908     --
909     ws3 = wb.create_sheet('Diffusion Results')
910     cols3 = ['Scenario', 'RL Backbone', 'Diffusion Model',
911             'Final Loss (mean±std)', 'Final Loss', 'Loss Std',
912             'Min Loss', 'Epochs', 'Num Trajectories',
913             'Loss Reduction %', 'Convergence Ep', 'Last-5 Avg Loss']
914     widths3 = [20, 14, 20, 26, 14, 12, 12, 10, 18, 16, 16, 16]
915     for c, (h, w) in enumerate(zip(cols3, widths3), 1):
916         hdr(ws3, 1, c, h, w)
917     ws3.freeze_panes = 'A2'
918     ws3.auto_filter.ref = ws3.dimensions
919
920     row = 2
921     for scenario in scenarios:
922         di_m = all_results[scenario].get('diffusion_models', {})
923         diff_d = statistical_results.get(scenario,
924                                         {}).get('diff_descriptive', {})
925         for model_key, d in di_m.items():
926             parts = model_key.split('_', 1)
927             rl_algo = parts[0]
928             diff_algo = parts[1] if len(parts) > 1 else model_key
929             desc = diff_d.get(model_key, {})
930             losses = d.get('losses', [])
931             last5 = float(np.mean(losses[-5:])) if losses else
932             float('nan')
933             fill = _ALT if row % 2 == 0 else None
934             fl = desc.get('final_loss', float('nan'))
935             ls = desc.get('loss_std', 0.0)
936             summary = (f"{nan_r(fl,6)} ± {nan_r(ls,6)}"
937                       if not (isinstance(fl, float) and np.isnan(fl))
938                       else '---')

```

```

935         dc(ws3, row, 1, scenario.replace('_', ' ').title(), fill)
936         dc(ws3, row, 2, rl_algo, fill)
937         dc(ws3, row, 3, diff_algo, fill)
938         dc(ws3, row, 4, summary, fill)
939         dc(ws3, row, 5, nan_r(fl, 6), fill,
FMT_6DP)
940         dc(ws3, row, 6, nan_r(ls, 6), fill,
FMT_6DP)
941         dc(ws3, row, 7, nan_r(desc.get('min_loss'), 6), fill,
FMT_6DP)
942         dc(ws3, row, 8, desc.get('epochs', len(losses)), fill)
943         dc(ws3, row, 9, desc.get('num_trajectories', 'N/A'), fill)
944         dc(ws3, row, 10, nan_r(desc.get('loss_reduction'), 2), fill,
FMT_2DP)
945         dc(ws3, row, 11, desc.get('convergence_ep', '---'), fill)
946         dc(ws3, row, 12, nan_r(last5, 6), fill,
FMT_6DP)
947         row += 1
948
949     # -- Sheet 4: Baselines -----
950     --
951     ws4 = wb.create_sheet('Baselines')
952     cols4 = ['Scenario', 'Random Cost', 'Greedy Cost', 'Hungarian Cost',
953             'CBBA Cost', 'Best Baseline', 'Best Cost', 'Worst Baseline',
954             'Worst Cost', 'CBBA vs Hungarian (%)', 'Greedy vs Hungarian
955             (%)']
956     widths4 = [22, 14, 14, 16, 14, 16, 14, 16, 14, 22, 22]
957     for c, (h, w) in enumerate(zip(cols4, widths4), 1):
958         hdr(ws4, 1, c, h, w)
959     ws4.freeze_panes = 'A2'
960     ws4.auto_filter.ref = ws4.dimensions
961
962     row = 2
963     for scenario in scenarios:

```



```

962     bc      = all_results[scenario].get('baseline_comparison', {})
963     costs = {n: bc.get(n, {}).get('avg_cost',
964                                   bc.get(n, {}).get('total_cost', float('nan'))))
965              for n in base_names}
966     valid = {k: v for k, v in costs.items() if not np.isnan(v)}
967     best  = min(valid, key=valid.get) if valid else '---'
968     worst = max(valid, key=valid.get) if valid else '---'
969     fill  = _ALT if row % 2 == 0 else None
970     dc(ws4, row, 1, scenario.replace('_', ' ').title(), fill,
bold=True)
971     for ci, bn in enumerate(base_names, 2):
972         dc(ws4, row, ci, nan_r(costs[bn], 2), fill, FMT_2DP)
973     dc(ws4, row, 6, best.title(), fill, bold=True)
974     dc(ws4, row, 7, nan_r(valid.get(best), 2), fill, FMT_2DP)
975     dc(ws4, row, 8, worst.title(), fill)
976     dc(ws4, row, 9, nan_r(valid.get(worst), 2), fill, FMT_2DP)
977     hung = costs['hungarian']; cbba = costs['cbba']; grdy =
costs['greedy']
978     dc(ws4, row, 10,
979         nan_r((cbba - hung) / (abs(hung) + 1e-9) * 100, 1)
980         if not (np.isnan(cbba) or np.isnan(hung)) else '---',
981         fill, FMT_2DP)
982     dc(ws4, row, 11,
983         nan_r((grdy - hung) / (abs(hung) + 1e-9) * 100, 1)
984         if not (np.isnan(grdy) or np.isnan(hung)) else '---',
985         fill, FMT_2DP)
986     row += 1
987
988     # -- Sheet 5: Statistical Tests --- RL pairwise -----
989     ----
990     ws5 = wb.create_sheet('Stat Tests (RL)')
991     note = ('Yellow = significant at  $\alpha=0.05$  (uncorrected) | '
992           'Green = Bonferroni-significant | Red = not Bonferroni-sig | '
993           'Purple = large effect ( $|d| \geq 0.8$ )')

```

```

993 ws5.cell(row=1, column=1, value=note).font = Font(italic=True, size=9,
994                                                    color='555555')
995 ws5.merge_cells(start_row=1, start_column=1, end_row=1, end_column=18)
996
997 cols5 = ['Scenario',
998          'Algorithm A', 'Mean A', 'Std A', '95% CI A',
999          'Algorithm B', 'Mean B', 'Std B', '95% CI B',
1000          't-statistic', 't p-value', 't Sig ( $\alpha=0.05$ )',
1001          'U-statistic', 'U p-value', 'U Sig ( $\alpha=0.05$ )',
1002          "Cohen's d", 'Effect Size', 'Bonferroni Sig']
1003 widths5 = [20, 14, 12, 10, 20, 14, 12, 10, 20,
1004             12, 12, 14, 12, 12, 14, 12, 12, 15]
1005 for c, (h, w) in enumerate(zip(cols5, widths5), 1):
1006     hdr(ws5, 2, c, h, w)
1007 ws5.freeze_panes = 'A3'
1008
1009 row = 3
1010 for scenario in scenarios:
1011     pairs = statistical_results.get(scenario, {}).get('rl_pairwise',
1012     [])
1013     for p in pairs:
1014         bf = p.get('bonferroni_sig')
1015         tp = p.get('t_pval', float('nan'))
1016         up = p.get('u_pval', float('nan'))
1017         cd = p.get('cohens_d', float('nan'))
1018         eff = p.get('effect_label', '')
1019         fill = _LARGE if eff == 'large' else sig_fill(tp, bf)
1020         dc(ws5, row, 1, scenario.replace('_', ' ').title(), fill)
1021         dc(ws5, row, 2, p.get('algo1', ''), fill, bold=True)
1022         dc(ws5, row, 3, nan_r(p.get('mean1'), 4), fill, FMT_4DP)
1023         dc(ws5, row, 4, nan_r(p.get('std1'), 4), fill, FMT_4DP)
1024         dc(ws5, row, 5,
            f"[{nan_r(p.get('ci1_low'),3)},
            {nan_r(p.get('ci1_high'),3)}]", fill)

```

```

1025         dc(ws5, row, 6, p.get('algo2', ''), fill, bold=True)
1026         dc(ws5, row, 7, nan_r(p.get('mean2'), 4), fill, FMT_4DP)
1027         dc(ws5, row, 8, nan_r(p.get('std2'), 4), fill, FMT_4DP)
1028         dc(ws5, row, 9,
1029             f"[{nan_r(p.get('ci2_low'),3)},
{nan_r(p.get('ci2_high'),3)}]", fill)
1030         dc(ws5, row, 10, nan_r(p.get('t_stat'), 4), fill, FMT_4DP)
1031         dc(ws5, row, 11, nan_r(tp, 6), fill, FMT_SCI)
1032         dc(ws5, row, 12, 'tick' if p.get('t_sig_raw') else 'cross',
fill,
1033             bold=p.get('t_sig_raw'))
1034         dc(ws5, row, 13, nan_r(p.get('u_stat'), 2), fill, FMT_2DP)
1035         dc(ws5, row, 14, nan_r(up, 6), fill, FMT_SCI)
1036         dc(ws5, row, 15, 'tick' if p.get('u_sig_raw') else 'cross',
fill,
1037             bold=p.get('u_sig_raw'))
1038         dc(ws5, row, 16, nan_r(cd, 4), fill, FMT_4DP)
1039         dc(ws5, row, 17, eff.capitalize(), eff_fill(eff))
1040         dc(ws5, row, 18, 'tick' if bf else 'cross', _BEST if
bf else _WARN,
1041             bold=bool(bf))
1042         row += 1
1043
1044     # -- Sheet 6: Statistical Tests --- Diffusion pairwise -----
1045     ----
1046     ws6 = wb.create_sheet('Stat Tests (Diffusion)')
1047     ws6.cell(row=1, column=1, value=note).font = Font(italic=True, size=9,
color='555555')
1048     ws6.merge_cells(start_row=1, start_column=1, end_row=1, end_column=13)
1049
1050     cols6 = ['Scenario',
1051             'Model A', 'Mean Loss A', 'Std A',
1052             'Model B', 'Mean Loss B', 'Std B',
1053             't-statistic', 't p-value', 't Sig ( $\alpha=0.05$ )',

```

```

1054         "Cohen's d", 'Effect Size', 'Bonferroni Sig']
1055     widths6 = [20, 24, 14, 12, 24, 14, 12, 12, 12, 14, 12, 14, 15]
1056     for c, (h, w) in enumerate(zip(cols6, widths6), 1):
1057         hdr(ws6, 2, c, h, w)
1058     ws6.freeze_panes = 'A3'
1059
1060     row = 3
1061     for scenario in scenarios:
1062         pairs = statistical_results.get(scenario,
1063         {}).get('diff_pairwise', [])
1064         for p in pairs:
1065             bf = p.get('bonferroni_sig')
1066             tp = p.get('t_pval', float('nan'))
1067             cd = p.get('cohens_d', float('nan'))
1068             eff = p.get('effect_label', '')
1069             fill = _LARGE if eff == 'large' else sig_fill(tp, bf)
1070             dc(ws6, row, 1, scenario.replace('_', ' ').title(), fill)
1071             dc(ws6, row, 2, p.get('model1', ''), fill, bold=True)
1072             dc(ws6, row, 3, nan_r(p.get('mean1'), 6), fill, FMT_6DP)
1073             dc(ws6, row, 4, nan_r(p.get('std1'), 6), fill, FMT_6DP)
1074             dc(ws6, row, 5, p.get('model2', ''), fill, bold=True)
1075             dc(ws6, row, 6, nan_r(p.get('mean2'), 6), fill, FMT_6DP)
1076             dc(ws6, row, 7, nan_r(p.get('std2'), 6), fill, FMT_6DP)
1077             dc(ws6, row, 8, nan_r(p.get('t_stat'), 4), fill, FMT_4DP)
1078             dc(ws6, row, 9, nan_r(tp, 6), fill, FMT_SCI)
1079             dc(ws6, row, 10, 'tick' if p.get('t_sig_raw') else 'cross',
1080             fill,
1081             bold=p.get('t_sig_raw'))
1082             dc(ws6, row, 11, nan_r(cd, 4), fill, FMT_4DP)
1083             dc(ws6, row, 12, eff.capitalize(), eff_fill(eff))
1084             dc(ws6, row, 13, 'tick' if bf else 'cross', _BEST if
1085             bf else _WARN,
1086             bold=bool(bf))
1087         row += 1

```

```

1085
1086 # -- Sheet 7: RL vs Baselines -----
1087
1088 ws7 = wb.create_sheet('RL vs Baselines')
1089 cols7 = ['Scenario', 'RL Algorithm',
1090          'RL Mean Reward', 'RL Completion Rate',
1091          'Hungarian Comp.', 'vs Hungarian (%)',
1092          'CBBA Comp.', 'vs CBBA (%)',
1093          'Greedy Comp.', 'vs Greedy (%)',
1094          'Random Comp.', 'vs Random (%)',
1095          'Beats All Baselines?']
1096 widths7 = [20, 14, 16, 18, 18, 18, 14, 16, 14, 16, 14, 16, 20]
1097 for c, (h, w) in enumerate(zip(cols7, widths7), 1):
1098     hdr(ws7, 1, c, h, w)
1099 ws7.freeze_panes = 'A2'
1100 ws7.auto_filter.ref = ws7.dimensions
1101
1102 row = 2
1103 for scenario in scenarios:
1104     rl_m = all_results[scenario].get('rl_models', {})
1105     rl_vb = statistical_results.get(scenario,
1106                                     {}).get('rl_vs_baseline', [])
1107     for entry in rl_vb:
1108         algo = entry['algo']
1109         fill = _BEST if algo == max(rl_m, key=lambda k:
1110                                     rl_m[k]['avg_reward'],
1111                                     default='') else (_ALT if row % 2
1112                                                         == 0 else None)
1113         dc(ws7, row, 1, scenario.replace('_', ' ').title(), fill)
1114         dc(ws7, row, 2, algo, fill, bold=True)
1115         dc(ws7, row, 3, nan_r(entry.get('rl_mean_reward'), 4),
1116            fill, FMT_4DP)
1117         dc(ws7, row, 4, nan_r(entry.get('rl_mean_completion'), 4),
1118            fill, FMT_4DP)

```

```

1113         beats = []
1114         for bn, col in [('hungarian', 5), ('cbba', 7), ('greedy', 9),
1115             ('random', 11)]:
1116             base_comp = entry.get(f'{bn}_completion', float('nan'))
1117             pct        = entry.get(f'{bn}_pct_improvement',
1118                 float('nan'))
1119             dc(ws7, row, col,
1120                 nan_r(base_comp, 4) if not np.isnan(base_comp) else
1121                 '---',
1122                 fill, FMT_4DP)
1123             pct_fill = (_BEST if (not np.isnan(pct) and pct > 0) else
1124                 _WARN if (not np.isnan(pct) and pct < 0) else
1125                 fill)
1126             dc(ws7, row, col + 1,
1127                 nan_r(pct, 1) if not np.isnan(pct) else '---',
1128                 pct_fill, FMT_2DP)
1129             if not np.isnan(pct) and pct > 0:
1130                 beats.append(bn)
1131             dc(ws7, row, 13,
1132                 'tick All' if len(beats) == 4 else
1133                 ('', '.join(beats) if beats else 'cross None'),
1134                 _BEST if len(beats) == 4 else fill,
1135                 bold=(len(beats) == 4))
1136             row += 1
1137
1138 # -- Sheet 8: Cohen's d Effect Size Matrix -----
1139 --
1140 ws8 = wb.create_sheet("Effect Sizes (RL)")
1141 ws8.cell(row=1, column=1,
1142         value=("Cohen's d matrix (row vs column). "
1143             "|d|<0.2 negligible, 0.2--0.5 small, 0.5--0.8 medium,
1144             ≥0.8 large.")
1145         ).font = Font(italic=True, size=9, color='555555')
1146 ws8.merge_cells(start_row=1, start_column=1,

```

```

1141         end_row=1, end_column=len(RL_ALGOS) + 3)
1142
1143     base_row = 2
1144     for scenario in scenarios:
1145         pairs = statistical_results.get(scenario, {}).get('rl_pairwise',
1146             [])
1147         d_map = {(p['algo1'], p['algo2']): p.get('cohens_d',
1148             float('nan'))
1149             for p in pairs}
1150         d_map.update({(p['algo2'], p['algo1']): -p.get('cohens_d',
1151             float('nan'))
1152             for p in pairs})
1153         ws8.cell(row=base_row, column=1,
1154             value=scenario.replace('_', ' ').title()).font = _BFONT
1155         base_row += 1
1156         hdr(ws8, base_row, 1, 'Algorithm \\ vs', 22, sub=True)
1157         for ci, algo in enumerate(RL_ALGOS, 2):
1158             hdr(ws8, base_row, ci, algo, 14, sub=True)
1159             base_row += 1
1160             for algo in RL_ALGOS:
1161                 dc(ws8, base_row, 1, algo, bold=True)
1162                 for ci, algo2 in enumerate(RL_ALGOS, 2):
1163                     if algo == algo2:
1164                         c = ws8.cell(row=base_row, column=ci, value='---')
1165                         c.fill = _ALT; c.alignment = _CTR; c.border = _BDR
1166                     else:
1167                         d = d_map.get((algo, algo2), float('nan'))
1168                         fill = (_LARGE if abs(d) >= 0.8 else
1169                             _BEST if abs(d) >= 0.5 else
1170                             _SIG if abs(d) >= 0.2 else None) \
1171                             if not np.isnan(d) else None
1172                         dc(ws8, base_row, ci, nan_r(d, 3), fill, FMT_4DP)
1173                     base_row += 1
1174             base_row += 1

```

```

1172
1173     # -- Sheet 9: Convergence Analysis -----
1174     --
1175     ws9 = wb.create_sheet('Convergence Analysis')
1176     cols9 = ['Scenario', 'RL Backbone', 'Diffusion Model',
1177             'Num Epochs', 'Num Trajectories',
1178             'Initial Loss', 'Final Loss (mean)', 'Min Loss',
1179             'Loss Reduction (%)', 'Convergence Episode',
1180             'Loss Std (final 10)', 'Training Stability']
1181     widths9 = [20, 14, 20, 12, 18, 14, 18, 14, 18, 20, 20, 18]
1182     for c, (h, w) in enumerate(zip(cols9, widths9), 1):
1183         hdr(ws9, 1, c, h, w)
1184     ws9.freeze_panes = 'A2'
1185     ws9.auto_filter.ref = ws9.dimensions
1186
1187     row = 2
1188     for scenario in scenarios:
1189         di_m = all_results[scenario].get('diffusion_models', {})
1190         diff_d = statistical_results.get(scenario,
1191         {}).get('diff_descriptive', {})
1192         for model_key, d in di_m.items():
1193             parts = model_key.split('_', 1)
1194             rl_algo = parts[0]
1195             diff_algo = parts[1] if len(parts) > 1 else model_key
1196             desc = diff_d.get(model_key, {})
1197             losses = d.get('losses', [])
1198             init = float(losses[0]) if losses else float('nan')
1199             fill = _ALT if row % 2 == 0 else None
1200             fl = desc.get('final_loss', float('nan'))
1201             ls = desc.get('loss_std', 0.0)
1202             stab = ('Stable' if (not np.isnan(ls) and not np.isnan(fl)
1203                 and fl > 0 and ls / fl < 0.05) else
1204                 'Unstable' if (not np.isnan(ls) and not np.isnan(fl)
1205                 and fl > 0 and ls / fl > 0.20) else

```



```

'Moderate')
1204         dc(ws9, row, 1, scenario.replace('_', ' ').title(), fill)
1205         dc(ws9, row, 2, rl_algo, fill)
1206         dc(ws9, row, 3, diff_algo, fill)
1207         dc(ws9, row, 4, desc.get('epochs', len(losses)), fill)
1208         dc(ws9, row, 5, desc.get('num_trajectories', 'N/A'), fill)
1209         dc(ws9, row, 6, nan_r(init, 6), fill,
FMT_6DP)
1210         dc(ws9, row, 7, nan_r(fl, 6), fill,
FMT_6DP)
1211         dc(ws9, row, 8, nan_r(desc.get('min_loss'), 6), fill,
FMT_6DP)
1212         dc(ws9, row, 9, nan_r(desc.get('loss_reduction'), 2), fill,
FMT_2DP)
1213         dc(ws9, row, 10, desc.get('convergence_ep', '---'), fill)
1214         dc(ws9, row, 11, nan_r(ls, 6), fill,
FMT_6DP)
1215         stab_fill = (_BEST if stab == 'Stable' else
1216                     _WARN if stab == 'Unstable' else _ALT)
1217         dc(ws9, row, 12, stab, stab_fill, bold=(stab == 'Stable'))
1218         row += 1
1219
1220     # -- Sheet 10: Energy Efficiency & Dynamic Reallocation -----
1221     --
1222     ws10 = wb.create_sheet('Energy & Reallocation')
1223     cols10 = [
1224         'Scenario', 'RL Algorithm',
1225         'Avg Energy/Ep (Wh)', 'Std Energy (Wh)',
1226         'Min Energy (Wh)', 'Max Energy (Wh)',
1227         'Avg Energy/Task (Wh)', 'Energy Efficiency (tasks/Wh)',
1228         'Avg Reallocations/Ep', 'Std Reallocations',
1229         'Total Reallocations', 'Realloc Rate (per 100 steps)',
1230     ]
1231     widths10 = [20, 14, 20, 18, 16, 16, 22, 28, 22, 18, 20, 26]

```

```

1231     for c, (h, w) in enumerate(zip(cols10, widths10), 1):
1232         hdr(ws10, 1, c, h, w)
1233     ws10.freeze_panes = 'A2'
1234     ws10.auto_filter.ref = ws10.dimensions
1235
1236     row10 = 2
1237     for scenario in scenarios:
1238         rl_m = all_results[scenario].get('rl_models', {})
1239         for algo in RL_ALGOS:
1240             if algo not in rl_m: continue
1241             d = rl_m[algo]
1242             fill = (_BEST if algo == max(rl_m,
1243                                     key=lambda k:
1244                                     rl_m[k].get('avg_energy_efficiency', 0),
1245                                     default=''))
1246             else (_ALT if row10 % 2 == 0 else None))
1247             energies = np.array(d.get('energies', [float('nan')])),
1248             dtype=float)
1249             energies = energies[~np.isnan(energies)]
1250             reallocs = np.array(d.get('reallocation_counts', [0])),
1251             dtype=float)
1252             comps = np.array(d.get('completions', [0]),
1253             dtype=float)
1254             avg_e = float(np.mean(energies)) if len(energies) else
1255             float('nan')
1256             std_e = float(np.std(energies, ddof=1)) if len(energies) > 1
1257             else 0.0
1258             min_e = float(np.min(energies)) if len(energies) else
1259             float('nan')
1260             max_e = float(np.max(energies)) if len(energies) else
1261             float('nan')
1262             n_tasks_ep = d.get('avg_completion', 0) * MAX_TASKS
1263             avg_e_per_task = ((avg_e / max(n_tasks_ep, 0.01))
1264                             if not np.isnan(avg_e) else float('nan'))

```

```

1257         eff    = d.get('avg_energy_efficiency', float('nan'))
1258         avg_r   = float(np.mean(reallocs)) if len(reallocs) else 0.0
1259         std_r   = float(np.std(reallocs, ddof=1)) if len(reallocs) > 1
        else 0.0
1260         tot_r   = int(np.sum(reallocs))
1261         n_ep    = len(reallocs)
1262         r_rate  = (tot_r / max(n_ep * MAX_STEPS, 1)) * 100
1263         dc(ws10, row10, 1,  scenario.replace('_', ' ').title(), fill,
        bold=True)
1264         dc(ws10, row10, 2,  algo,                                fill, bold=True)
1265         dc(ws10, row10, 3,  nan_r(avg_e, 4),                    fill, FMT_4DP)
1266         dc(ws10, row10, 4,  nan_r(std_e, 4),                    fill, FMT_4DP)
1267         dc(ws10, row10, 5,  nan_r(min_e, 4),                    fill, FMT_4DP)
1268         dc(ws10, row10, 6,  nan_r(max_e, 4),                    fill, FMT_4DP)
1269         dc(ws10, row10, 7,  nan_r(avg_e_per_task, 4), fill, FMT_4DP)
1270         dc(ws10, row10, 8,  nan_r(eff, 6),                      fill, FMT_6DP)
1271         dc(ws10, row10, 9,  nan_r(avg_r, 2),                    fill, FMT_2DP)
1272         dc(ws10, row10, 10, nan_r(std_r, 2),                    fill, FMT_2DP)
1273         dc(ws10, row10, 11, tot_r,                                fill)
1274         dc(ws10, row10, 12, nan_r(r_rate, 3),                    fill, FMT_4DP)
1275         row10 += 1
1276
1277     wb.save(filepath)
1278     logger.info('\nResults exported to Excel: %s', filepath)
1279     logger.info('  Sheets (%d): %s', len(wb.sheetnames), ',
        '.join(wb.sheetnames))
1280     return filepath
1281
1282
1283 # -- Ablation constants (paths are constructed lazily inside functions) --
        -----
1284
1285     _RESULTS_DIR      = 'results'
1286     _ABLATION_SUBDIR  = 'ablation'

```

```

1287 _ABLATION_SCENARIO = 'warehouse'
1288 _ABLATION_RL_EP     = 500
1289 _ABLATION_DIFF_EP   = 50
1290 _ABLATION_TRAJ_EP    = 100

```

Listing B.10: Diagnostic and analysis utilities for the environment package.

Package: models

```

1 # models/__init__.py

```

Listing B.11: Package init for the models package.

```

1 """
2 models/attention.py
3 =====
4 Attention-based observation encoder, adapted from DCMRTA (ICRA 2024).
5
6 Treats the flat observation as structured tokens rather than a single
7     vector:
8     - Robot tokens: (MAX_ROBOTS, robot_feat_dim)
9     - Task tokens:  (MAX_TASKS,  task_feat_dim)
10    - Global stats: (1, stats_dim)
11
12 Multi-head attention across tokens lets the encoder learn agent-task
13 affinity directly, which reduces sample complexity for sparse-reward MARL
14 compared to a flat MLP baseline.
15
16 Architecture details:
17     - GateFFN with sigmoid gating (more stable than ReLU in low-data
18       regimes).
19     - tanh output clipping at  $\pm 10$  prevents pointer-network logit explosion.
20     - LayerNorm after every attention block for training stability.

```

```

19     - Encoder-decoder cross-attention captures bidirectional agent-task
      context.
20 """
21 import torch
22 import torch.nn as nn
23 import math
24 import numpy as np
25 from typing import Final, Optional, Tuple
26
27 from config.base_config import (
28     MAX_ROBOTS, MAX_TASKS, GLOBAL_OBS_DIM,
29     GLOBAL_ROBOT_DIM, GLOBAL_TASK_DIM, GLOBAL_STATS_DIM,
30     FLAT_OBS_DIM, FLAT_ROBOT_DIM, FLAT_TASK_DIM,
31     FLAT_TASK_FEATS, FLAT_ROBOT_FEATS
32 )
33
34
35 # GateFFN (DCMRTA: sigmoid gating improves gradient flow)
36
37 class GateFFN(nn.Module):
38     """
39     Gated feed-forward network.
40     h = sigmoid(W1(x)) * V1(x)
41     out = W2(h)
42     Better gradient flow than standard ReLU FFN.
43     """
44     def __init__(self, model_dim: int, hidden_dim: int = None):
45         super().__init__()
46         hidden_dim = hidden_dim or model_dim * 4
47         self.W = nn.Linear(model_dim, hidden_dim, bias=False)
48         self.V = nn.Linear(model_dim, hidden_dim, bias=False)
49         self.W2 = nn.Linear(hidden_dim, model_dim, bias=False)
50         self.act = nn.Sigmoid()
51

```

```

52     def forward(self, x: torch.Tensor) -> torch.Tensor:
53         return self.W2(self.act(self.W(x)) * self.V(x))
54
55
56 class TransformerBlock(nn.Module):
57     """Single transformer block: MHA residual LayerNorm GateFFN residual
58         LayerNorm."""
59
60     def __init__(self, embed_dim: int, n_heads: int):
61         super().__init__()
62         self.attn = nn.MultiheadAttention(embed_dim, n_heads,
63             batch_first=True)
64         self.norm1 = nn.LayerNorm(embed_dim)
65         self.ffn = GateFFN(embed_dim)
66         self.norm2 = nn.LayerNorm(embed_dim)
67
68     def forward(self, x: torch.Tensor,
69         kv: Optional[torch.Tensor] = None,
70         key_padding_mask: Optional[torch.Tensor] = None) ->
71         torch.Tensor:
72         if kv is None:
73             kv = x
74         attn_out, _ = self.attn(x, kv, kv,
75             key_padding_mask=key_padding_mask)
76         x = self.norm1(x + attn_out)
77         x = self.norm2(x + self.ffn(x))
78         return x
79
80 class SingleHeadPointer(nn.Module):
81     """
82     Pointer network (DCMRTA SingleHeadAttention).
83     Scores each token in a set and returns log-probabilities.
84     tanh clipping prevents logit explosion.

```

```

82 """
83     def __init__(self, embed_dim: int, tanh_clip: float = 10.0):
84         super().__init__()
85         self.tanh_clip = tanh_clip
86         self.W_q = nn.Parameter(torch.Tensor(embed_dim, embed_dim))
87         self.W_k = nn.Parameter(torch.Tensor(embed_dim, embed_dim))
88         self.norm_factor = 1.0 / math.sqrt(embed_dim)
89         nn.init.uniform_(self.W_q, -1/math.sqrt(embed_dim),
90             1/math.sqrt(embed_dim))
91         nn.init.uniform_(self.W_k, -1/math.sqrt(embed_dim),
92             1/math.sqrt(embed_dim))
93
94     def forward(self, query: torch.Tensor,
95         keys: torch.Tensor,
96         mask: Optional[torch.Tensor] = None) -> torch.Tensor:
97         """
98         Args:
99             query: (batch, 1, embed_dim)
100             keys: (batch, n_items, embed_dim)
101             mask: (batch, n_items) bool, True = invalid
102
103         Returns:
104             log_probs: (batch, n_items)
105         """
106         Q = torch.matmul(query, self.W_q) # (batch, 1, embed_dim)
107         K = torch.matmul(keys, self.W_k) # (batch, n_items, embed_dim)
108         u = self.norm_factor * torch.matmul(Q, K.transpose(-2, -1)) #
109             (batch, 1, n_items)
110         u = self.tanh_clip * torch.tanh(u).squeeze(1) # (batch, n_items)
111         if mask is not None:
112             u = u.masked_fill(mask, -1e8)
113         return torch.log_softmax(u, dim=-1)

```

```

113 # Main observation encoder
114
115 class ObservationEncoder(nn.Module):
116     """
117     Structured attention encoder for multi-robot task allocation.
118
119     Input: global_obs (batch, 470) - structured as [robots×8, tasks×6,
120         stats×10]
121
122     Output: encoded representation for use by RL policy head
123
124     Architecture:
125     1. Project robot/task tokens to embed_dim
126     2. Encode robot tokens with self-attention (agent context)
127     3. Encode task tokens with self-attention (task context)
128     4. Cross-decode: each robot attends to all tasks
129     5. Mean-pool tasks global context
130     6. Final representation: concatenation of [robot_enc, global_ctx]
131
132     """
133
134     # TorchScript requires integer constants to be Final class attributes,
135     # not closed-over module globals.
136     MAX_ROBOTS: Final[int] = 20
137     MAX_TASKS: Final[int] = 50
138
139     def __init__(self, embed_dim: int = 128, n_heads: int = 8,
140         n_layers: int = 2):
141         super().__init__()
142         self.embed_dim = embed_dim
143
144         # Robot token: 8 features (pos×2, vel×2, battery, payload,
145         status, has_task)
146         robot_feat = 8
147
148         # Task token: 6 features (pos×2, priority, payload_norm,
149         deadline_norm, status)

```



```

144     task_feat = 6
145     # Stats: 10 features
146     stats_feat = 10
147
148     self.robot_embed = nn.Linear(robot_feat, embed_dim)
149     self.task_embed = nn.Linear(task_feat, embed_dim)
150     self.stats_embed = nn.Linear(stats_feat, embed_dim)
151
152     # Self-attention encoders
153     self.robot_encoder = nn.ModuleList([
154         TransformerBlock(embed_dim, n_heads) for _ in
range(n_layers)])
155     self.task_encoder = nn.ModuleList([
156         TransformerBlock(embed_dim, n_heads) for _ in
range(n_layers)])
157
158     # Cross-attention: robots attend to tasks
159     self.cross_decoder = nn.ModuleList([
160         TransformerBlock(embed_dim, n_heads) for _ in
range(n_layers)])
161
162     # Output projection
163     self.out_proj = nn.Linear(embed_dim + embed_dim, embed_dim)
164
165     def _parse_global_obs(self, obs: torch.Tensor) -> Tuple[torch.Tensor,
torch.Tensor, torch.Tensor]:
166         """Split global_obs into structured token tensors.
167
168         Layout: [MAX_ROBOTS×8 = 160, MAX_TASKS×6 = 300, stats×10 = 10] =
470 dims.
169
170         Inactive robot/task slots are zero-padded by the observation
extractor.
171
172         If obs is shorter than 470, pad with zeros so reshapes never
crash.

```

```

171     """
172     batch = obs.shape[0]
173     D      = obs.shape[-1]
174     if D < 470:
175         obs = torch.cat([obs, obs.new_zeros(batch, 470 - D)], dim=-1)
176     robots = obs[:, :160].reshape(batch, self.MAX_ROBOTS, 8)
177     tasks  = obs[:, 160:460].reshape(batch, self.MAX_TASKS, 6)
178     stats  = obs[:, 460:470]
179     return robots, tasks, stats
180
181     def forward(self, obs: torch.Tensor,
182                 num_robots: int = 0,
183                 num_tasks: int = 0) -> torch.Tensor:
184         """Encode global observation into a rich representation.
185
186         Args:
187             obs:          (batch, 470) global observation (inactive slots
zero-padded)
188             num_robots:   active fleet size this episode; 0 = use MAX_ROBOTS
189             num_tasks:    active task count this episode; 0 = use MAX_TASKS
190
191         Returns:
192             encoded: (batch, embed_dim)
193
194         """
195         n_r = num_robots if num_robots > 0 else self.MAX_ROBOTS
196         n_t = num_tasks  if num_tasks  > 0 else self.MAX_TASKS
197
198         robots_raw, tasks_raw, stats_raw = self._parse_global_obs(obs)
199
200         # Embed all token positions (same weights regardless of N)
201         R = self.robot_embed(robots_raw)    # (batch, MAX_ROBOTS,
embed_dim)
202         T = self.task_embed(tasks_raw)      # (batch, MAX_TASKS,
embed_dim)

```

```

202         S = self.stats_embed(stats_raw).unsqueeze(1)  # (batch, 1,
           embed_dim)
203
204         # Self-attention over robots
205         for layer in self.robot_encoder:
206             R = layer(R)
207
208         # Self-attention over tasks
209         for layer in self.task_encoder:
210             T = layer(T)
211
212         # Cross-attention: robots attend to tasks
213         RT = R
214         for layer in self.cross_decoder:
215             RT = layer(RT, kv=T)  # (batch, MAX_ROBOTS, embed_dim)
216
217         # Pool only over active agents/tasks --- inactive zero-padded
           slots excluded
218         task_ctx = T[:, :n_t, :].mean(dim=1) + S.squeeze(1)  # (batch,
           embed_dim)
219         robot_ctx = RT[:, :n_r, :].mean(dim=1)  # (batch,
           embed_dim)
220
221         out = self.out_proj(torch.cat([robot_ctx, task_ctx], dim=-1))
222         return out  # (batch, embed_dim)
223
224
225 class TaskPointerHead(nn.Module):
226     """
227     Pointer network head for direct task selection.
228     Takes encoded robot/task representations and scores each task.
229     Returns log-probabilities over tasks.
230     """
231

```

```

232     # TorchScript requires integer constants to be Final class attributes.
233     MAX_ROBOTS: Final[int] = 20
234     MAX_TASKS: Final[int] = 50
235
236     def __init__(self, embed_dim: int = 128, n_heads: int = 8, tanh_clip:
        float = 10.0):
237         super().__init__()
238         self.robot_embed = nn.Linear(8, embed_dim)
239         self.task_embed = nn.Linear(6, embed_dim)
240         self.robot_encoder = TransformerBlock(embed_dim, n_heads)
241         self.task_encoder = TransformerBlock(embed_dim, n_heads)
242         self.pointer = SingleHeadPointer(embed_dim, tanh_clip)
243
244     def forward(self, obs: torch.Tensor,
245                 mask: Optional[torch.Tensor] = None,
246                 num_robots: int = 0,
247                 num_tasks: int = 0) -> torch.Tensor:
248         """Args:
249             obs:          (batch, 470) global obs
250             mask:          (batch, MAX_TASKS) bool, True = invalid task
251             num_robots:    active fleet size; 0 = MAX_ROBOTS
252             num_tasks:     active task count; 0 = MAX_TASKS
253         Returns:
254             log_probs: (batch, MAX_TASKS)
255         """
256         batch = obs.shape[0]
257         n_r = num_robots if num_robots > 0 else self.MAX_ROBOTS
258         robots_raw = obs[:, :160].reshape(batch, self.MAX_ROBOTS, 8)
259         tasks_raw = obs[:, 160:460].reshape(batch, self.MAX_TASKS, 6)
260
261         R = self.robot_encoder(self.robot_embed(robots_raw))
262         T = self.task_encoder(self.task_embed(tasks_raw))
263
264         # Query = mean over active robots only

```

```

265         query = R[:, :n_r, :].mean(dim=1, keepdim=True)    # (batch, 1,
        embed_dim)
266
267         return self.pointer(query, T, mask=mask)    # (batch, MAX_TASKS)

```

Listing B.12: Attention-based observation encoder shared across all five RL agents (DCMRTA-derived).

```

1  """
2  models/rl_agents.py
3  =====
4  Five RL agent classes: MAPPO, QMIX, MADDPG, DQN, HybridRL.
5
6  Shared architecture across all five agents:
7      - Attention-based observation encoder (DCMRTA) replaces flat MLP input.
8      - GateFFN hidden layers (sigmoid gating) instead of standard FFN.
9      - tanh output clipping at  $\pm 10$  stabilises pointer-network logits.
10     - Invalid action masking applied before every softmax; repeated at eval.
11     - All classes are TorchScript-compatible for .pt export to simulation.
12
13  HybridRL selects among the four base agents using UCB bandit selection,
14  distilling from pre-trained Stage A checkpoints to warm-start.
15  """
16  import torch
17  import torch.nn as nn
18
19
20  def il_demon_rate(episode: int,
21                   initial_rate: float = 0.5,
22                   decay: float = -1e-5) -> float:
23      """
24      Decaying IL demon rate (DCMRTA parameters.py).
25      DEMON_RATE = 0.5, IL_DECAY = -1e-5
26      => decays from 0.5 to ~0 over 70k episodes.

```

```

27     Formula: max(0, initial_rate + decay * episode)
28     """
29     return max(0.0, initial_rate + decay * episode)
30 import torch.nn.functional as F
31 import numpy as np
32 from typing import Final, Optional, Tuple, Dict
33
34 from config.base_config import (
35     MAX_ROBOTS, MAX_TASKS, ACTION_DIM, LOCAL_OBS_DIM, GLOBAL_OBS_DIM,
36     FLAT_OBS_DIM, RLConfig
37 )
38 from models.attention import ObservationEncoder, GateFFN, TransformerBlock
39 from utils.action_masking import mask_logits, masked_softmax
40
41
42 cfg = RLConfig()
43
44 # Shared components
45
46 class MLP(nn.Module):
47     """Simple MLP with GateFFN hidden layers."""
48     def __init__(self, in_dim: int, out_dim: int, hidden_dim: int = 256,
49                 n_layers: int = 2):
50         super().__init__()
51         layers = [nn.Linear(in_dim, hidden_dim),
52                 nn.LayerNorm(hidden_dim), nn.ReLU()]
53         for _ in range(n_layers - 1):
54             layers += [nn.Linear(hidden_dim, hidden_dim),
55                     nn.LayerNorm(hidden_dim), nn.ReLU()]
56         layers.append(nn.Linear(hidden_dim, out_dim))
57         self.net = nn.Sequential(*layers)
58
59     def forward(self, x):
60         return self.net(x)

```

```

58
59
60 class AttentionActorHead(nn.Module):
61     """
62     Actor head using attention encoder + pointer network.
63     Input: local_obs (149) or global_obs (470)
64     Output: log_probs over action_dim
65     """
66     def __init__(self, obs_dim: int, action_dim: int,
67                 embed_dim: int = 128, hidden_dim: int = 256,
68                 tanh_clip: float = 10.0):
69         super().__init__()
70         if obs_dim == GLOBAL_OBS_DIM:
71             self.encoder = ObservationEncoder(embed_dim)
72             enc_out = embed_dim
73         else:
74             # Local obs: simple MLP encoder
75             self.encoder = MLP(obs_dim, embed_dim, hidden_dim)
76             enc_out = embed_dim
77
78         self.head = nn.Linear(enc_out, action_dim)
79         self.tanh_clip = tanh_clip
80
81     def forward(self, obs: torch.Tensor,
82                mask: Optional[torch.Tensor] = None) -> torch.Tensor:
83         enc = self.encoder(obs)
84         logits = self.head(enc)
85         # tanh clipping (DCMRTA)
86         logits = self.tanh_clip * torch.tanh(logits / self.tanh_clip)
87         if mask is not None:
88             logits = mask_logits(logits, mask)
89         return logits
90
91

```

```

92 class ValueHead(nn.Module):
93     def __init__(self, obs_dim: int, hidden_dim: int = 256, embed_dim:
94         int = 128):
95         super().__init__()
96         if obs_dim == GLOBAL_OBS_DIM:
97             self.encoder = ObservationEncoder(embed_dim)
98             in_dim = embed_dim
99         else:
100             self.encoder = MLP(obs_dim, embed_dim, hidden_dim)
101             in_dim = embed_dim
102         self.head = nn.Linear(in_dim, 1)
103
104     def forward(self, obs: torch.Tensor) -> torch.Tensor:
105         return self.head(self.encoder(obs))
106
107 # 1. MAPPO
108
109 class MAPPOActor(nn.Module):
110     def __init__(self, local_obs_dim: int = LOCAL_OBS_DIM,
111         action_dim: int = ACTION_DIM,
112         hidden_dim: int = cfg.hidden_dim,
113         embed_dim: int = cfg.embed_dim):
114         super().__init__()
115         self.actor = AttentionActorHead(local_obs_dim, action_dim,
116             embed_dim, hidden_dim)
117
118     def forward(self, local_obs: torch.Tensor,
119         mask: Optional[torch.Tensor] = None) -> torch.Tensor:
120         return self.actor(local_obs, mask)
121
122 class MAPPOCritic(nn.Module):
123     def __init__(self, global_obs_dim: int = GLOBAL_OBS_DIM,

```



```

124         hidden_dim: int = cfg.hidden_dim,
125         embed_dim: int = cfg.embed_dim):
126     super().__init__()
127     self.critic = ValueHead(global_obs_dim, hidden_dim, embed_dim)
128
129     def forward(self, global_obs: torch.Tensor) -> torch.Tensor:
130         return self.critic(global_obs)
131
132
133 class MAPPO(nn.Module):
134     """
135     Multi-Agent PPO with attention encoder, action masking, and centralised
136     critic.
137     """
138     name = 'MAPPO'
139     MAX_TASKS: Final[int] = 50
140
141     def __init__(self, local_obs_dim=LOCAL_OBS_DIM,
142                  global_obs_dim=GLOBAL_OBS_DIM,
143                  action_dim=ACTION_DIM, hidden_dim=cfg.hidden_dim,
144                  embed_dim=cfg.embed_dim):
145         super().__init__()
146         self.actor = MAPPOActor(local_obs_dim, action_dim, hidden_dim,
147                                 embed_dim)
148         self.critic = MAPPOCritic(global_obs_dim, hidden_dim, embed_dim)
149         self.action_dim = action_dim
150
151     def get_action(self, local_obs: torch.Tensor, global_obs:
152                   torch.Tensor,
153                   mask: Optional[torch.Tensor] = None,
154                   deterministic: bool = False) -> Tuple[torch.Tensor,
155                   torch.Tensor, torch.Tensor]:
156         logits = self.actor(local_obs, mask)
157         probs = torch.softmax(logits, dim=-1)

```

```

152         dist = torch.distributions.Categorical(probs=probs)
153         if deterministic:
154             action = torch.argmax(probs, dim=-1)
155         else:
156             action = dist.sample()
157         log_prob = dist.log_prob(action)
158         value = self.critic(global_obs).squeeze(-1)
159         return action, log_prob, value
160
161     def evaluate(self, local_obs, global_obs, actions, mask=None):
162         logits = self.actor(local_obs, mask)
163         probs = torch.softmax(logits, dim=-1)
164         dist = torch.distributions.Categorical(probs=probs)
165         log_prob = dist.log_prob(actions)
166         entropy = dist.entropy()
167         value = self.critic(global_obs).squeeze(-1)
168         return log_prob, entropy, value
169
170     def forward(self, local_obs: torch.Tensor,
171                 flat_obs: Optional[torch.Tensor] = None,
172                 mask: Optional[torch.Tensor] = None) ->
173         Tuple[torch.Tensor, torch.Tensor]:
174         """TorchScript-compatible forward for sim export.
175         flat_obs is accepted but unused; present so all RL models share the
176         same two-arg signature and SimReadyRL can be scripted with any model."""
177         logits = self.actor(local_obs, mask)
178         max_tasks: int = self.MAX_TASKS
179         action = torch.argmax(logits, dim=-1)
180         robot_id = action // max_tasks
181         task_id = action % max_tasks
182         return robot_id, task_id
183
184 # 2. QMIX

```

```

185
186 class QMIXAgent(nn.Module):
187     """Individual Q-network for QMIX, with attention encoder."""
188     def __init__(self, local_obs_dim=LOCAL_OBS_DIM, action_dim=ACTION_DIM,
189                 hidden_dim=cfg.hidden_dim, embed_dim=cfg.embed_dim):
190         super().__init__()
191         self.encoder = MLP(local_obs_dim, embed_dim, hidden_dim)
192         self.q_head = nn.Linear(embed_dim, action_dim)
193         self.tanh_clip = cfg.tanh_clip
194
195     def forward(self, local_obs: torch.Tensor,
196                mask: Optional[torch.Tensor] = None) -> torch.Tensor:
197         enc = self.encoder(local_obs)
198         q = self.q_head(enc)
199         q = self.tanh_clip * torch.tanh(q / self.tanh_clip)
200         if mask is not None:
201             q = mask_logits(q, mask)
202         return q
203
204
205 class QMIXMixer(nn.Module):
206     """Monotonic mixing network --- supports dynamic fleet size up to
207     max_agents.
208
209     Hypernetworks are sized for MAX_ROBOTS at construction; at forward
210     time
211     only the first n_active rows are used, so the same weights handle any
212     fleet size from 1 to max_agents without reloading the model.
213     """
214     def __init__(self, n_agents: int = MAX_ROBOTS, state_dim: int =
215                 GLOBAL_OBS_DIM,
216                 mixing_hidden: int = 32):
217         super().__init__()
218         self.n_agents = n_agents # maximum fleet size (weight dim)

```

```

216     self.mixing_hidden = mixing_hidden
217     # Hypernetworks generate positive weights; output covers max fleet
218     self.hyper_w1 = nn.Sequential(
219         nn.Linear(state_dim, mixing_hidden),
220         nn.ReLU(),
221         nn.Linear(mixing_hidden, n_agents * mixing_hidden))
222     self.hyper_b1 = nn.Linear(state_dim, mixing_hidden)
223     self.hyper_w2 = nn.Sequential(
224         nn.Linear(state_dim, mixing_hidden),
225         nn.ReLU(),
226         nn.Linear(mixing_hidden, mixing_hidden))
227     self.hyper_b2 = nn.Sequential(
228         nn.Linear(state_dim, mixing_hidden),
229         nn.ReLU(),
230         nn.Linear(mixing_hidden, 1))
231     # Small output-layer init prevents q_total explosion at training
start.
232     # Default kaiming init gives w1≈0.5 → bmm ≈
n_agents×10×0.5=100 →
233     # q_total≈3200, causing loss≈420 and grad≈3630 at ep=200 which
cascades
234     # to NaN by ep=400 via the AMP float16 range limit.
235     nn.init.normal_(self.hyper_w1[-1].weight, std=0.01)
236     nn.init.zeros_(self.hyper_w1[-1].bias)
237     nn.init.normal_(self.hyper_w2[-1].weight, std=0.01)
238     nn.init.zeros_(self.hyper_w2[-1].bias)
239
240     def forward(self, q_vals: torch.Tensor, state: torch.Tensor) ->
        torch.Tensor:
241         """Args:
242             q_vals: (batch, n_active) --- Q values for the active fleet
        this episode
243             state: (batch, state_dim) --- global state
244         Returns:

```

```

245         q_total: (batch, 1)
246         """
247         batch = q_vals.shape[0]
248         n_active = q_vals.shape[1] # actual fleet size --- may be <
self.n_agents
249
250         # Generate weight matrix for max fleet, then slice to active
agents
251         w1 = torch.abs(self.hyper_w1(state)).reshape(
252             batch, self.n_agents, self.mixing_hidden)
253         w1 = w1[:, :n_active, :] # (batch, n_active, mh)
254
255         b1 = self.hyper_b1(state).unsqueeze(1) # (batch, 1,
mixing_hidden)
256         h = F.elu(torch.bmm(q_vals.unsqueeze(1), w1) + b1) # (batch,
1, mh)
257
258         w2 = torch.abs(self.hyper_w2(state)).reshape(batch,
259             self.mixing_hidden, 1)
260         b2 = self.hyper_b2(state)
261         q_total = (torch.bmm(h, w2).squeeze(-1) + b2)
262         return q_total
263
264 class QMIX(nn.Module):
265     """QMIX with a single shared QMIXAgent applied to every robot.
266
267     The same weights handle any fleet size 1--MAX_ROBOTS: call
268     get_agent_q()
269     for each robot in a loop; the mixer slices its weight matrix to the
270     active fleet count at forward time. No per-index heads.
271     """
272     name = 'QMIX'
273     MAX_TASKS: Final[int] = 50

```

```

273
274 def __init__(self, local_obs_dim=LOCAL_OBS_DIM,
275             global_obs_dim=GLOBAL_OBS_DIM,
276             action_dim=ACTION_DIM, n_agents=MAX_ROBOTS,
277             hidden_dim=cfg.hidden_dim, embed_dim=cfg.embed_dim):
278     super().__init__()
279     # Single shared agent --- same weights for every robot regardless
280     of fleet size
281     self.shared_agent = QMIXAgent(local_obs_dim, action_dim,
282     hidden_dim, embed_dim)
283
284     self.mixer = QMIXMixer(n_agents, global_obs_dim)
285
286     self.n_agents = n_agents
287
288     self.action_dim = action_dim
289
290
291 def get_agent_q(self, local_obs: torch.Tensor,
292               agent_id: int,
293               mask: Optional[torch.Tensor] = None) -> torch.Tensor:
294     """Returns Q-values for one robot using the shared network."""
295     return self.shared_agent(local_obs, mask)
296
297
298 def forward(self, local_obs: torch.Tensor,
299             flat_obs: Optional[torch.Tensor] = None,
300             mask: Optional[torch.Tensor] = None) ->
301     Tuple[torch.Tensor, torch.Tensor]:
302     """TorchScript-compatible forward for sim.
303     flat_obs is accepted but unused; present so all RL models share
304     the
305     same two-arg signature and SimReadyRL can be scripted with any
306     model."""
307
308     q = self.shared_agent(local_obs, mask)
309
310     max_tasks: int = self.MAX_TASKS
311
312     action = torch.argmax(q, dim=-1)
313
314     robot_id = action // max_tasks
315
316     task_id = action % max_tasks

```

```

301         return robot_id, task_id
302
303
304     # 3. MADDPG (discrete Gumbel-softmax variant)
305     # switched from plain softmax-Categorical to Gumbel-softmax sampling so
306     # the actor can produce differentiable one-hot actions that the
307         centralised
308     # critic can backprop through - this is the standard discrete-MADDPG trick
309     # (Lowe et al. 2017 §4.2, reused by e.g. MARLlib and PettingZoo's discrete
310     # MADDPG). Temperature is annealed from MADDPG_GUMBEL_TEMP_START (1.0) to
311     # MADDPG_GUMBEL_TEMP_END (0.1) over MADDPG_GUMBEL_TEMP_DECAY episodes
312         (5000
313     # by default, see environment.env_config.RLConfig).
314     #
315     # LIMITATION / TODO: the centralised-critic DPG actor update that fully
316     # justifies the Gumbel path (maximise Q_critic(s, actor_gumbel(s))) is not
317     # yet wired into training/rl_trainer.py - still trains the actor with a
318     # TD-style loss on the sampled action. The Gumbel-softmax change here is
319     # prerequisite plumbing for that; finishing the DPG update is a ~50-line
320     # addition in rl_trainer.py and is noted for post-FYP work.
321
322     class MADDPGActor(nn.Module):
323
324         def __init__(self, obs_dim=FLAT_OBS_DIM, action_dim=ACTION_DIM,
325                     hidden_dim=cfg.hidden_dim, embed_dim=cfg.embed_dim):
326             super().__init__()
327             self.encoder = (ObservationEncoder(embed_dim) if obs_dim ==
328                             GLOBAL_OBS_DIM
329                             else MLP(obs_dim, embed_dim, hidden_dim))
330             self.actor = nn.Linear(embed_dim, action_dim)
331             self.tanh_clip = cfg.tanh_clip
332
333         def logits(self, obs: torch.Tensor,
334                   mask: Optional[torch.Tensor] = None) -> torch.Tensor:
335             enc = self.encoder(obs)

```

```

332         a = self.tanh_clip * torch.tanh(self.actor(enc) / self.tanh_clip)
333         if mask is not None:
334             a = mask_logits(a, mask)
335         return a
336
337     def forward(self, obs: torch.Tensor,
338                 mask: Optional[torch.Tensor] = None) -> torch.Tensor:
339         """Return raw logits (kept for backward compatibility)."""
340         return self.logits(obs, mask)
341
342     def gumbel_sample(self, obs: torch.Tensor,
343                      tau: float = 1.0,
344                      hard: bool = True,
345                      mask: Optional[torch.Tensor] = None) ->
346         torch.Tensor:
347         """
348         Differentiable one-hot action via Gumbel-softmax with straight-through
349         estimator. Returns shape (B, action_dim). Use hard=True during training
350         so the forward pass sees a true one-hot and the backward pass gets the
351         soft-softmax gradient.
352         """
353         lg = self.logits(obs, mask)
354         return F.gumbel_softmax(lg, tau=tau, hard=hard, dim=-1)
355
356     class MADDPGCritic(nn.Module):
357         def __init__(self, obs_dim=FLAT_OBS_DIM, action_dim=ACTION_DIM,
358                     hidden_dim=cfg.hidden_dim):
359             super().__init__()
360             self.net = MLP(obs_dim + action_dim, 1, hidden_dim)
361
362         def forward(self, obs: torch.Tensor, action_onehot: torch.Tensor) ->
363             torch.Tensor:
364             return self.net(torch.cat([obs, action_onehot], dim=-1))

```



```

364
365
366 class MADDPG(nn.Module):
367     name = 'MADDPG'
368     MAX_TASKS: Final[int] = 50
369     FLAT_OBS_DIM: Final[int] = 910
370
371     def __init__(self, obs_dim=FLAT_OBS_DIM, action_dim=ACTION_DIM,
372                 hidden_dim=cfg.hidden_dim, embed_dim=cfg.embed_dim):
373         super().__init__()
374         self.actor = MADDPGActor(obs_dim, action_dim, hidden_dim,
375                                 embed_dim)
376         self.critic = MADDPGCritic(obs_dim, action_dim, hidden_dim)
377         self.action_dim = action_dim
378         # Gumbel-softmax temperature (annealed by training loop via
379         # step_gumbel_temp(episode)). Registered as a buffer so it's
380         saved in
381         # state_dict and restored on resume.
382         self.register_buffer('gumbel_tau', torch.tensor(1.0))
383
384     def step_gumbel_temp(self, episode: int,
385                        tau_start: float = 1.0,
386                        tau_end: float = 0.1,
387                        decay_eps: int = 5000) -> float:
388         """Linearly anneal Gumbel-softmax temperature over decay_eps
389         episodes."""
390         frac = min(1.0, max(0.0, episode / max(1, decay_eps)))
391         tau = tau_start + frac * (tau_end - tau_start)
392         self.gumbel_tau.fill_(tau)
393         return tau
394
395     def get_action(self, flat_obs: torch.Tensor,
396                  mask: Optional[torch.Tensor] = None,
397                  deterministic: bool = False) -> torch.Tensor:

```

```

395         """
396         Sample a discrete action index.
397         - deterministic=True -> argmax over masked logits (inference)
398         - deterministic=False -> Gumbel-softmax straight-through sample
399         (training; gradient flows through actor)
400         """
401         if deterministic:
402             with torch.no_grad():
403                 lg = self.actor.logits(flat_obs, mask)
404                 return torch.argmax(lg, dim=-1)
405         # Training: gumbel-softmax with current annealed temperature
406         tau = float(self.gumbel_tau.item())
407         onehot = self.actor.gumbel_sample(flat_obs, tau=tau, hard=True,
408                                           mask=mask)
409         # Return the integer index for the sampled action. argmax of a
410         hard
411         # one-hot gives back the sampled index without breaking the
412         # straight-through gradient path for any downstream critic use.
413         return torch.argmax(onehot, dim=-1)
414
415     def forward(self, obs: torch.Tensor,
416                flat_obs: Optional[torch.Tensor] = None,
417                mask: Optional[torch.Tensor] = None) ->
418         Tuple[torch.Tensor, torch.Tensor]:
419         """TorchScript-compatible forward for sim. obs can be local(149)
420         or flat(910).
421         flat_obs is accepted but unused; present so all RL models share the
422         same two-arg signature and SimReadyRL can be scripted with any model."""
423         if obs.shape[-1] < self.FLAT_OBS_DIM:
424             pad = obs.new_zeros(obs.shape[0], self.FLAT_OBS_DIM -
425                                obs.shape[-1])
426             obs = torch.cat([obs, pad], dim=-1)
427         logits = self.actor.logits(obs, mask)
428         max_tasks: int = self.MAX_TASKS

```

```

424         action = torch.argmax(logits, dim=-1)
425         robot_id = action // max_tasks
426         task_id = action % max_tasks
427         return robot_id, task_id
428
429
430 # 4. DQN (Dueling + Double)
431 # upgraded from vanilla DQN to Dueling + Double DQN. env_config.RLConfig
432 # already advertised USE_DOUBLE_DQN=True / USE_DUELING_DQN=True but the
433     class
434 # had only an online/target linear head, so those flags were no-ops that
435 # lied about the implementation. Now honoured:
436 # - Dueling:  $Q(s,a) = V(s) + A(s,a) - \text{mean}_a A(s,a)$  (Wang et al. 2016)
437 # - Double:  $\text{target} = r + \gamma * Q_{\text{target}}(s', \text{argmax}_a Q_{\text{online}}(s', a))$ 
438 # (van Hasselt et al. 2016). The target-action-selection lives in
439 # the training loop (rl_trainer.py), not the model; the model
440 # exposes both q_online and q_target via q_values(use_target=...).
441 # Prioritised Experience Replay (PER) is separate: see utils (new PER
442     buffer
443     # would live there) - wiring into rl_trainer is a bigger change and is
444     left
445     # as a TODO for post-FYP polish. For now USE_PER=True in env_config is
446     still
447     # a placeholder flag; the PER buffer in RLConfig.buffer_size / per_alpha /
448     # per_beta_start is allocated but not yet sampled-from in the loop.
449
450
451 class DQN(nn.Module):
452     """Dueling Double DQN."""
453     name = 'DQN'
454     MAX_TASKS: Final[int] = 50
455     FLAT_OBS_DIM: Final[int] = 910
456
457     def __init__(self, obs_dim=FLAT_OBS_DIM, action_dim=ACTION_DIM,
458                 hidden_dim=cfg.hidden_dim, embed_dim=cfg.embed_dim):

```

```

454     super().__init__()
455     self.encoder = MLP(obs_dim, embed_dim, hidden_dim)
456
457     # Dueling heads: value stream V(s) and advantage stream A(s,a)
458     # for BOTH online and target networks.
459     self.v_online = nn.Linear(embed_dim, 1)
460     self.a_online = nn.Linear(embed_dim, action_dim)
461     self.v_target = nn.Linear(embed_dim, 1)
462     self.a_target = nn.Linear(embed_dim, action_dim)
463
464     self.action_dim = action_dim
465     self.tanh_clip = cfg.tanh_clip
466     # Copy online weights to target at init
467     self.v_target.load_state_dict(self.v_online.state_dict())
468     self.a_target.load_state_dict(self.a_online.state_dict())
469
470     def _dueling_q(self, enc: torch.Tensor, use_target: bool) ->
471         torch.Tensor:
472         if use_target:
473             v = self.v_target(enc)                # (B, 1)
474             a = self.a_target(enc)                # (B, A)
475         else:
476             v = self.v_online(enc)
477             a = self.a_online(enc)
478
479             # Dueling aggregation:  $Q = V + (A - \text{mean}_a A)$ , mean-subtracted to
480             make
481
482             # the decomposition identifiable (Wang et al. 2016 eq. 9).
483             q = v + (a - a.mean(dim=-1, keepdim=True))
484
485             # tanh clip same as
486             q = self.tanh_clip * torch.tanh(q / self.tanh_clip)
487         return q
488
489     def q_values(self, flat_obs: torch.Tensor,
490                 mask: Optional[torch.Tensor] = None,

```

```

486         use_target: bool = False) -> torch.Tensor:
487     enc = self.encoder(flat_obs)
488     q = self._dueling_q(enc, use_target=use_target)
489     if mask is not None:
490         q = mask_logits(q, mask)
491     return q
492
493     def get_action(self, flat_obs: torch.Tensor,
494                   eps: float = 0.0,
495                   mask: Optional[torch.Tensor] = None) -> torch.Tensor:
496         if np.random.random() < eps:
497             valid = (~mask).nonzero(as_tuple=True)[0] if mask is not None\
498                 else torch.arange(self.action_dim)
499             return valid[torch.randint(len(valid), (1,))].squeeze()
500         with torch.no_grad():
501             q = self.q_values(flat_obs, mask)
502             return torch.argmax(q, dim=-1)
503
504     def update_target(self):
505         self.v_target.load_state_dict(self.v_online.state_dict())
506         self.a_target.load_state_dict(self.a_online.state_dict())
507
508     def forward(self, obs: torch.Tensor,
509                flat_obs: Optional[torch.Tensor] = None,
510                mask: Optional[torch.Tensor] = None) ->
511        Tuple[torch.Tensor, torch.Tensor]:
512        """TorchScript-compatible forward for sim.
513        flat_obs is accepted but unused; present so all RL models share the
514        same two-arg signature and SimReadyRL can be scripted with any model."""
515        if obs.shape[-1] < self.FLAT_OBS_DIM:
516            pad = obs.new_zeros(obs.shape[0], self.FLAT_OBS_DIM -
517                                obs.shape[-1])
518            obs = torch.cat([obs, pad], dim=-1)
519            q = self.q_values(obs, mask)

```

```

518         max_tasks: int = self.MAX_TASKS
519         action    = torch.argmax(q, dim=-1)
520         robot_id = action // max_tasks
521         task_id  = action % max_tasks
522         return robot_id, task_id
523
524
525 # 5. HybridRL
526
527 class HybridRL(nn.Module):
528     """
529     HybridRL: ensemble of ALL 4 base RL algorithms (MAPPO, QMIX, MADDPG, DQN)
530     with UCB bandit selection at each step.
531
532     UCB bandit tracks cumulative reward per algorithm and selects the one
533     with the highest upper confidence bound - automatically discovering
534     which algorithm works best per scenario as training progresses.
535
536     At inference (sim): the algorithm with the highest historical reward is
537     used deterministically.
538     """
539     name = 'HybridRL'
540     N_ALGOS = 4    # MAPPO, QMIX, MADDPG, DQN
541     MAX_TASKS: Final[int] = 50
542     GLOBAL_OBS_DIM: Final[int] = 470
543     FLAT_OBS_DIM: Final[int] = 910    # must match
544         base_config.FLAT_OBS_DIM
545
546     def __init__(self, local_obs_dim=LOCAL_OBS_DIM,
547         flat_obs_dim=FLAT_OBS_DIM,
548         action_dim=ACTION_DIM, hidden_dim=cfg.hidden_dim,
549         embed_dim=cfg.embed_dim):
550         super().__init__()

```

```

550     # All 4 base RL algorithms as sub-networks
551     self.mappo_actor = MAPPOActor(local_obs_dim, action_dim,
hidden_dim, embed_dim)
552     self.mappo_critic = MAPPOCritic(GLOBAL_OBS_DIM, hidden_dim,
embed_dim)
553     self.qmix_agent = QMIXAgent(local_obs_dim, action_dim,
hidden_dim, embed_dim)
554     self.maddpg_actor = MADDPGActor(flat_obs_dim, action_dim,
hidden_dim, embed_dim)
555     self.dqn_encoder = MLP(flat_obs_dim, embed_dim, hidden_dim)
556     self.dqn_head = nn.Linear(embed_dim, action_dim)
557
558     self.action_dim = action_dim
559     self.tanh_clip = cfg.tanh_clip
560
561     # UCB bandit: one counter per algorithm [MAPPO, QMIX, MADDPG, DQN]
562     self.register_buffer('algo_counts', torch.ones(self.N_ALGOS))
563     self.register_buffer('algo_rewards', torch.zeros(self.N_ALGOS))
564     self.register_buffer('algo_wins', torch.zeros(self.N_ALGOS))
565     # Gumbel-softmax temperature annealed by rl_trainer for the MADDPG
566     # sub-actor inside HybridRL. Declared here (not dynamically via
567     # register_buffer in the trainer) so the key is always present in
568     # state_dict and Stage-C checkpoint loads never fail with
569     # "Unexpected key(s) in state_dict: gumbel_tau".
570     self.register_buffer('gumbel_tau', torch.tensor(1.0))
571
572     def _ucb_select(self, total_steps: int) -> int:
573         """UCB1 bandit: select algo with highest upper confidence
bound."""
574         if total_steps < self.N_ALGOS:
575             return total_steps % self.N_ALGOS
576         avg = self.algo_rewards / self.algo_counts
577         ucb = avg + torch.sqrt(
578             2 * torch.log(torch.tensor(float(total_steps + 1))) /

```

```

self.algo_counts)
579     return int(torch.argmax(ucb).item())
580
581 def _run_algo(self, algo_id: int,
582             local_obs: torch.Tensor, flat_obs: torch.Tensor,
583             mask: Optional[torch.Tensor], deterministic: bool
584             ) -> Tuple[torch.Tensor, torch.Tensor]:
585     """Run a single sub-algorithm, return (action, log_prob)."""
586     if algo_id == 0:  # MAPPO
587         logits = self.mappo_actor(local_obs, mask)
588         probs = torch.softmax(logits, dim=-1)
589         dist = torch.distributions.Categorical(probs)
590         action = torch.argmax(probs, dim=-1) if deterministic else
dist.sample()
591         lp = dist.log_prob(action)
592
593     elif algo_id == 1:  # QMIX
594         q = self.qmix_agent(local_obs, mask)
595         probs = torch.softmax(q, dim=-1)
596         dist = torch.distributions.Categorical(probs)
597         action = torch.argmax(probs, dim=-1) if deterministic else
dist.sample()
598         lp = dist.log_prob(action)
599
600     elif algo_id == 2:  # MADDPG
601         logits = self.maddpg_actor(flat_obs, mask)
602         probs = torch.softmax(logits, dim=-1)
603         dist = torch.distributions.Categorical(probs)
604         action = torch.argmax(probs, dim=-1) if deterministic else
dist.sample()
605         lp = dist.log_prob(action)
606
607     else:  # DQN
608         q = self.tanh_clip * torch.tanh(

```



```

609         self.dqn_head(self.dqn_encoder(flat_obs)) /
self.tanh_clip)
610         if mask is not None:
611             q = mask_logits(q, mask)
612             action = torch.argmax(q, dim=-1)
613             lp     = torch.zeros(action.shape, device=action.device)
614
615         return action, lp
616
617     def get_action(self, local_obs: torch.Tensor, flat_obs: torch.Tensor,
618                   mask: Optional[torch.Tensor] = None,
619                   deterministic: bool = False,
620                   step: int = 1) -> Tuple[torch.Tensor, torch.Tensor,
torch.Tensor, int]:
621         """
622         UCB bandit selects which sub-algorithm fires this step.
623         Returns (action, log_prob, value, algo_id_used).
624         """
625         algo_id = self._ucb_select(step) if not deterministic else\
626             int(torch.argmax(self.algo_wins).item())
627
628         action, log_prob = self._run_algo(algo_id, local_obs, flat_obs,
629                                           mask, deterministic)
630         value = self.mappo_critic(flat_obs[:, :self.GLOBAL_OBS_DIM]\
631                                   if flat_obs.shape[-1] >=
self.GLOBAL_OBS_DIM
632                                   else flat_obs).squeeze(-1)
633         return action, log_prob, value, algo_id
634
635     def evaluate(self, local_obs: torch.Tensor, global_obs: torch.Tensor,
636                 actions: torch.Tensor,
637                 mask: Optional[torch.Tensor] = None
638                 ) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
639         """PPO evaluation pass using the MAPPO sub-actor and critic."""

```

```

640         logits    = self.mappo_actor(local_obs, mask)
641         probs      = torch.softmax(logits, dim=-1)
642         dist       = torch.distributions.Categorical(probs=probs)
643         log_prob   = dist.log_prob(actions)
644         entropy    = dist.entropy()
645         value      = self.mappo_critic(global_obs).squeeze(-1)
646         return log_prob, entropy, value
647
648     def update_bandit(self, algo_id: int, reward: float, won: bool =
        False):
649         """Update UCB bandit statistics after an episode."""
650         self.algo_counts[algo_id] += 1
651         self.algo_rewards[algo_id] += reward
652         if won:
653             self.algo_wins[algo_id] += 1
654
655     def get_all_actions(self, local_obs: torch.Tensor, flat_obs:
        torch.Tensor,
656                        mask: Optional[torch.Tensor] = None
657                        ) -> Dict[int, torch.Tensor]:
658         """Run all 4 algorithms and return their actions - used for
        ablation."""
659         return {i: self._run_algo(i, local_obs, flat_obs, mask, True)[0]
660                for i in range(self.N_ALGOS)}
661
662     def forward(self, local_obs: torch.Tensor,
663                flat_obs: Optional[torch.Tensor] = None,
664                mask: Optional[torch.Tensor] = None) ->
        Tuple[torch.Tensor, torch.Tensor]:
665         """
666         TorchScript-compatible forward for sim.
667         Uses the historically best-performing algorithm (deterministic argmax).
668
669         Intentionally does NOT call _run_algo because that path goes through

```

```

670 torch.distributions.Categorical whose _new(**kwargs) signature is not
671 scriptable. We inline the four deterministic action selections here -
672 no sampling is needed at inference time.
673
674 flat_obs is Optional so SimReadyRL can be scripted uniformly across
675 all RL model types (non-HybridRL branches call self.model with 1 arg).
676 """
677     if flat_obs is None:
678         flat_obs = local_obs.new_zeros(local_obs.shape[0],
        self.FLAT_OBS_DIM)
679
680     best_algo = int(torch.argmax(self.algo_wins).item())
681
682     # Deterministic argmax for each sub-algorithm - no
torch.distributions used.
683     if best_algo == 0:                                     # MAPPO: actor logits
argmax
684         logits = self.mappo_actor(local_obs, mask)
685         action = torch.argmax(logits, dim=-1)
686     elif best_algo == 1:                                     # QMIX: Q-values
argmax
687         q_vals = self.qmix_agent(local_obs, mask)
688         action = torch.argmax(q_vals, dim=-1)
689     elif best_algo == 2:                                     # MADDPG: actor
logits argmax
690         logits = self.maddpg_actor(flat_obs, mask)
691         action = torch.argmax(logits, dim=-1)
692     else:                                                    # DQN: dueling Q
argmax
693         enc = self.dqn_encoder(flat_obs)
694         q_vals = self.tanh_clip * torch.tanh(self.dqn_head(enc) /
        self.tanh_clip)
695         if mask is not None:
696             q_vals = mask_logits(q_vals, mask)

```

```

697         action = torch.argmax(q_vals, dim=-1)
698
699         max_tasks: int = self.MAX_TASKS
700         robot_id = action // max_tasks
701         task_id = action % max_tasks
702         return robot_id, task_id
703
704
705 # Factory
706
707 def build_rl_agent(algo: str, **kwargs) -> nn.Module:
708     agents = {
709         'MAPPO': MAPPO,
710         'QMIX': QMIX,
711         'MADDPG': MADDPG,
712         'DQN': DQN,
713         'HybridRL': HybridRL,
714     }
715     if algo not in agents:
716         raise ValueError(f'Unknown RL algorithm: {algo}')
717     return agents[algo](**kwargs)

```

Listing B.13: All five RL agent classes: MAPPO, QMIX, MADDPG, DQN and HybridRL with shared encoder backbone and runtime UCB bandit selection.

```

1 """
2 models/diffusion_models.py
3 =====
4 Five diffusion trajectory model classes: TDM, DiffusionPolicy, Diffuser,
5 SDE and HybridDiffusion.
6
7 Shared design:
8 - B-spline representation: 8 control points → 20 dense waypoints via

```

```

9     BSplineLayer. Reduces output dimension and enforces smoothness
10     structurally (MPD-splines, Carvalho et al. T-RO 2025).
11     - Cosine noise schedule preserves higher SNR for longer than linear,
12     producing cleaner denoising on trajectory data.
13     - Goal conditioning on (start_pos, goal_pos) rather than RL action
14     one-hot;
15     cross-attention mode ('attention') gives stronger goal following.
16     - Cost-guided denoising: collision and smoothness gradients steer each
17     DDIM step without retraining (MPD-splines + MMD).
18     - DDIM deterministic sampling at inference (ddim_steps << T diffusion
19     steps).
20
21 HybridDiffusion ensembles TDM and SDE outputs with cost-guided weighting.
22 """
23 import torch
24 import torch.nn as nn
25 import torch.nn.functional as F
26 import numpy as np
27 from typing import Optional, Tuple, List
28
29 from config.base_config import (
30     MAX_ROBOTS, MAX_TASKS, GLOBAL_OBS_DIM, FLAT_OBS_DIM,
31     DiffusionConfig
32 )
33 from models.attention import GateFFN, TransformerBlock, ObservationEncoder
34 from utils.bspline import BSplineLayer, smoothness_loss,
35     inter_robot_collision_gradient
36 from utils.noise_schedule import DiffusionSchedule
37
38 cfg = DiffusionConfig()
39
40 # Shared score network (U-Net style for trajectory denoising)

```

```

40 class SinusoidalTimeEmbed(nn.Module):
41     """Sinusoidal timestep embedding."""
42     def __init__(self, dim: int):
43         super().__init__()
44         self.dim = dim
45         half = dim // 2
46         emb = torch.log(torch.tensor(10000.0)) / (half - 1)
47         self.register_buffer('freqs', torch.exp(-emb *
48             torch.arange(half)))
49
50     def forward(self, t: torch.Tensor) -> torch.Tensor:
51         """t: (batch,) integer timesteps (batch, dim)"""
52         t = t.float().unsqueeze(-1) * self.freqs.unsqueeze(0)
53         return torch.cat([torch.sin(t), torch.cos(t)], dim=-1)
54
55 class ScoreNet(nn.Module):
56     """
57     Score network: predicts noise  $\epsilon$  given (x_t, t, cond).
58
59     Supports two conditioning modes (MPD-splines TemporalUnet improvement
60     U+V):
61     - 'concatenate' (default): cond concatenated to input - simple, fast
62     - 'attention': goal encoded separately and injected via cross-attention
63     in each GateFFN block - stronger goal-following
64
65     Architecture improvements:
66     W: dim_mults=(1,2,4,8) inspired hierarchy via hidden_dim scaling
67     X: state_encoder MLP bottleneck reduces goal-cond dim before concat
68     """
69     def __init__(self, n_ctrl: int = cfg.bspline_n_ctrl,
70         cond_dim: int = None,
71         hidden_dim: int = cfg.hidden_dim,
72         time_embed_dim: int = cfg.time_embed_dim,

```

```

72         conditioning_type: str = 'attention'):
73     super().__init__()
74     self.n_ctrl          = n_ctrl
75     self.conditioning_type = conditioning_type
76     # TorchScript: forward cannot look up attributes on the
77     # DiffusionConfig dataclass (it is a plain Python value). Copy
78     # the handful of ints we actually need inside forward onto
79     # self at __init__ time so forward sees only pure tensors /
80     # self attributes / module calls.
81     self.goal_cond_dim: int = int(cfg.goal_cond_dim)
82     # TS-friendly mode flag: scripted forward branches on this int
83     # rather than doing string equality (marginally safer across
84     # torch versions and makes the intent explicit).
85     self._is_attention: int = 1 if conditioning_type == 'attention'
else 0
86     traj_flat = n_ctrl * 2
87     cond_dim  = cond_dim or (GLOBAL_OBS_DIM + cfg.goal_cond_dim)
88
89     self.time_embed = SinusoidalTimeEmbed(time_embed_dim)
90
91     # MPD-splines improvement X: state encoder bottleneck
92     # Embeds goal conditioning (4-dim) into a richer space before
injection
93     self.goal_encoder = nn.Sequential(
94         nn.Linear(cfg.goal_cond_dim, hidden_dim // 4),
95         nn.SiLU(),
96         nn.Linear(hidden_dim // 4, hidden_dim // 4),
97     )
98     goal_embed_dim = hidden_dim // 4
99
100    # Attention-path submodules (always registered)
101    # TorchScript compiles *both* branches of the if-else in forward,
102    # so every attribute it touches must exist at __init__ time even
if

```

```

103     # only one branch will execute at runtime. We pay a little extra
104     # memory for the unused branch but the model authoring stays
105     clean.
106     obs_dim_in = cond_dim - cfg.goal_cond_dim    # obs part only
107     self.obs_proj = nn.Linear(obs_dim_in, hidden_dim)
108     in_dim_attn = traj_flat + time_embed_dim
109     self.fc_in = nn.Linear(in_dim_attn, hidden_dim)
110     # Improvement W: 4-layer hierarchy (dim_mults 1,2,4,8 4 hidden
111     blocks)
112     self.block1 = nn.Sequential(nn.LayerNorm(hidden_dim), nn.SiLU(),
113     GateFFN(hidden_dim))
114     self.block2 = nn.Sequential(nn.LayerNorm(hidden_dim), nn.SiLU(),
115     GateFFN(hidden_dim))
116     self.block3 = nn.Sequential(nn.LayerNorm(hidden_dim), nn.SiLU(),
117     GateFFN(hidden_dim))
118     self.block4 = nn.Sequential(nn.LayerNorm(hidden_dim), nn.SiLU(),
119     GateFFN(hidden_dim))
120     # Cross-attention: query=hidden, key/value=goal_embed
121     self.cross_attn = nn.MultiheadAttention(
122         embed_dim=hidden_dim, num_heads=4,
123         kdim=goal_embed_dim, vdim=goal_embed_dim,
124         batch_first=True)
125     self.fc_out = nn.Linear(hidden_dim, traj_flat)
126
127     # Concatenate-path submodule (always registered)
128     in_dim_concat = traj_flat + time_embed_dim + cond_dim
129     self.net = nn.Sequential(
130         nn.Linear(in_dim_concat, hidden_dim),
131         nn.LayerNorm(hidden_dim), nn.SiLU(), GateFFN(hidden_dim),
132         nn.Linear(hidden_dim, hidden_dim),
133         nn.LayerNorm(hidden_dim), nn.SiLU(), GateFFN(hidden_dim),
134         nn.Linear(hidden_dim, hidden_dim),
135         nn.LayerNorm(hidden_dim), nn.SiLU(), GateFFN(hidden_dim),
136         nn.Linear(hidden_dim, traj_flat),

```



```

131         )
132
133     def forward(self, x_t: torch.Tensor, t: torch.Tensor,
134                 cond: torch.Tensor) -> torch.Tensor:
135         batch      = x_t.shape[0]
136         x_flat     = x_t.reshape(batch, -1)
137         t_emb      = self.time_embed(t)
138
139         if self._is_attention == 1:
140             # Split cond into obs part and goal part
141             goal_part = cond[:, -self.goal_cond_dim:]          # (batch, 4)
142             obs_part  = cond[:, :-self.goal_cond_dim]          # (batch,
143             obs_enc_dim)
144
145             # MPD-splines X: embed goal through bottleneck MLP
146             goal_emb = self.goal_encoder(goal_part)            # (batch,
147             hidden//4)
148             obs_ctx  = self.obs_proj(obs_part)                  # (batch,
149             hidden)
150
151             h = self.fc_in(torch.cat([x_flat, t_emb], dim=-1)) # (batch,
152             hidden)
153             h = h + obs_ctx                                     # fuse
154             obs
155
156             # MPD-splines W: 4-block hierarchy (like dim_mults 1,2,4,8
157             blocks)
158             h = h + self.block1(h)
159             h = h + self.block2(h)
160
161             # MPD-splines V: cross-attention - goal guides trajectory
162             refinement
163             # query: trajectory hidden state, key/value: goal embedding
164             goal_kv = goal_emb.unsqueeze(1)                    # (batch, 1,

```

```

158         h_q      = h.unsqueeze(1)                # (batch,
159         1, hidden)
160         h_attn, _ = self.cross_attn(h_q, goal_kv, goal_kv)
161         h = h + h_attn.squeeze(1)                # residual
162
163         h = h + self.block3(h)
164         h = h + self.block4(h)
165         eps = self.fc_out(h).reshape(batch, self.n_ctrl, 2)
166     else:
167         h = torch.cat([x_flat, t_emb, cond], dim=-1)
168         eps = self.net(h).reshape(batch, self.n_ctrl, 2)
169
170     return eps
171
172 class DiffusionBase(nn.Module):
173     """
174     Base class for all diffusion models.
175     Handles: B-spline layer, noise schedule, DDIM sampling,
176     cost-guided denoising, goal conditioning.
177     """
178
179     def __init__(self, name: str,
180                 obs_dim: int = GLOBAL_OBS_DIM,
181                 n_ctrl: int = cfg.bspline_n_ctrl,
182                 hidden_dim: int = cfg.hidden_dim,
183                 schedule: str = cfg.noise_schedule,
184                 T: int = cfg.T):
185         super().__init__()
186         self.model_name = name
187         self.n_ctrl      = n_ctrl
188         self.obs_dim     = obs_dim
189

```

```

190     # B-spline layer (MPD-splines)
191     self.bspline = BSplineLayer(n_ctrl, cfg.bspline_degree,
    cfg.bspline_n_eval)
192     # MMD improvement Q: velocity buffer refined during DDIM guidance
    steps
193     self._velocity_buf: Optional[torch.Tensor] = None
194
195     # Obs encoder
196     if obs_dim == GLOBAL_OBS_DIM:
197         self.obs_encoder = ObservationEncoder(cfg.embed_dim)
198         enc_out = cfg.embed_dim
199     else:
200         self.obs_encoder = nn.Sequential(
201             nn.Linear(obs_dim, hidden_dim), nn.LayerNorm(hidden_dim),
    nn.SiLU(),
202             nn.Linear(hidden_dim, cfg.embed_dim))
203         enc_out = cfg.embed_dim
204
205     # Goal conditioning: [enc_out + goal_cond_dim]
206     self.cond_dim = enc_out + cfg.goal_cond_dim
207     self.score_net = ScoreNet(n_ctrl, self.cond_dim, hidden_dim,
208
    conditioning_type=cfg.conditioning_type)
209
210     # Training-time noise schedule (lazy, device-aware)
211     # Used only by training_loss and.sample - both are marked
212     # @torch.jit.unused so TorchScript never sees this path.
213     self._schedule = None
214     self._schedule_name = schedule
215     self._T = T
216
217     # Scriptable inference path buffers
218     # _infer_traj runs a no-guidance DDIM loop and must be
219     # TorchScript-compilable. DiffusionSchedule is a plain Python

```

```

class,
220     # so we instead pre-compute the two tensors that DDIM actually
needs
221     # (alphas_cumprod and sqrt(1-alphas_cumprod)) and store them as
222     # registered buffers. register_buffer moves them with.to(device)
223     # automatically, same as in BSplineLayer.
224     _s = DiffusionSchedule(schedule, T, torch.device('cpu'))
225     self.register_buffer('_sc_acp',    _s.alphas_cumprod)
226     self.register_buffer('_sc_om_acp',
_s.sqrt_one_minus_alphas_cumprod)
227     # Plain int attributes TorchScript can read inside forward.
228     self._T_steps:    int = int(T)
229     self._ddim_steps: int = int(cfg.ddim_steps)
230
231     @torch.jit.unused
232     def _get_schedule(self, device) -> DiffusionSchedule:
233         if self._schedule is None or self._schedule.device != device:
234             self._schedule = DiffusionSchedule(self._schedule_name,
self._T, device)
235         return self._schedule
236
237     def _make_cond(self, obs: torch.Tensor,
238                    start_pos: torch.Tensor,
239                    goal_pos: torch.Tensor) -> torch.Tensor:
240         """
241         Build conditioning vector: [obs_encoding, start_x, start_y, goal_x,
goal_y]
242         Goal conditioning from MPD-splines: much richer than RL action one-hot.
243         """
244         enc    = self.obs_encoder(obs)           # (batch, embed_dim)
245         cond   = torch.cat([enc, start_pos, goal_pos], dim=-1) # (batch,
cond_dim)
246         return cond
247

```

```

248     @torch.jit.unused
249     def training_loss(self, traj_wpts: torch.Tensor,
250                       obs: torch.Tensor,
251                       start_pos: torch.Tensor,
252                       goal_pos: torch.Tensor) -> torch.Tensor:
253         """
254         Compute DDPM training loss with smoothness regularisation.
255
256         Args:
257         traj_wpts: (batch, n_eval, 2) - ground-truth trajectory waypoints
258         obs: (batch, obs_dim) - conditioning observation
259         start_pos: (batch, 2) - robot start position (normalised)
260         goal_pos: (batch, 2) - task goal position (normalised)
261
262         Returns:
263         loss: scalar
264         """
265         sched = self._get_schedule(obs.device)
266         batch = obs.shape[0]
267         device = obs.device
268
269         # Fit B-spline ctrl points to target trajectory
270         # We do this by least-squares using the precomputed basis matrix
271         B = sched.sqrt_alphas_cumprod.new_zeros(1) # just to get device
272         bspline = self.bspline
273         B_mat = bspline._get_basis(device) # (n_eval, n_ctrl)
274         # Pseudo-inverse: ctrl_pts = (B^T B)^{-1} B^T traj_wpts
275         BtB = B_mat.t().mm(B_mat) # (n_ctrl, n_ctrl)
276         Bt = B_mat.t() # (n_ctrl, n_eval)
277         # (batch, n_ctrl, 2) = (n_ctrl, n_eval) @ (batch, n_eval, 2)
278         # torch.linalg.solve requires float32 (lu_factor_cublas rejects
279         # float16).
280         # Casting the *inputs* to matmul is not enough when called inside
281         an AMP

```

```

280         # autocast region: AMP intercepts torch.matmul and runs it in
float16,
281         # producing a Half output even if the inputs were explicitly
float32.
282         # Cast the matmul *result* to float32 - tensor.float is a type
283         # cast, not an op, so autocast cannot override it.
284         _orig_dtype = traj_wpts.dtype
285         ctrl_pts = torch.linalg.solve(
286             BtB.unsqueeze(0).expand(batch, -1, -1).float(),
287             torch.matmul(Bt.unsqueeze(0).expand(batch, -1, -1).float(),
288                 traj_wpts.float()).float(), # cast result,
not inputs
289         ).to(_orig_dtype)
290
291         # Sample timestep
292         t = torch.randint(0, sched.T, (batch,), device=device)
293
294         # Forward process
295         noise = torch.randn_like(ctrl_pts)
296         x_noisy = sched.q_sample(ctrl_pts, t, noise)
297
298         # Predict noise
299         cond = self._make_cond(obs, start_pos, goal_pos)
300         eps_hat = self.score_net(x_noisy, t, cond)
301
302         # MSE noise loss
303         mse_loss = F.mse_loss(eps_hat, noise)
304
305         # Smoothness loss on denoised trajectory (MPD-splines)
306         # Reconstruct estimated x_0 and evaluate spline.
307         # Clamp before bspline evaluation: at high-noise timesteps,
predict_x0_from_eps
308         # amplifies x0_est by 1/sqrt(alpha_bar) which can be 10-30x,
making smoothness

```

```

309         # loss huge (~3000) and causing training divergence.
310         x0_est = sched.predict_x0_from_eps(x_noisy, t, eps_hat)
311         x0_est = x0_est.clamp(-5.0, 5.0)
312         traj_est = bspline(x0_est)          # (batch, n_eval, 2)
313         smooth_l = smoothness_loss(traj_est)
314
315         return mse_loss + cfg.smoothness_loss_weight * smooth_l
316
317     @torch.jit.unused
318     @torch.no_grad()
319     def sample(self, obs: torch.Tensor,
320               start_pos: torch.Tensor,
321               goal_pos: torch.Tensor,
322               n_samples: int = 1,
323               use_guidance: bool = True,
324               other_trajs: Optional[torch.Tensor] = None) ->
325         torch.Tensor:
326         """
327         DDIM sampling with optional cost guidance.
328
329         Args:
330         obs: (1, obs_dim) - conditioning observation
331         start_pos: (1, 2) - start position (normalised)
332         goal_pos: (1, 2) - goal position (normalised)
333         n_samples: number of trajectory samples to generate
334         use_guidance: apply collision + smoothness gradient guidance
335         other_trajs: (n_robots, n_eval, 2) trajectories of other robots (for MMD
336                     constraint)
337
338         Returns:
339         trajectories: (n_samples, n_eval, 2) in normalised coordinates
340         """
341         sched = self._get_schedule(obs.device)
342         device = obs.device

```

```

341
342     # Ensure obs is 2D (batch, dim)
343     if obs.dim() == 3: obs = obs.squeeze(1)
344     if start_pos.dim() == 3: start_pos = start_pos.squeeze(1)
345     if goal_pos.dim() == 3: goal_pos = goal_pos.squeeze(1)
346     obs_rep      = obs.expand(n_samples, -1)
347     start_rep    = start_pos.expand(n_samples, -1)
348     goal_rep     = goal_pos.expand(n_samples, -1)
349     cond        = self._make_cond(obs_rep, start_rep, goal_rep)
350
351     # Start from pure noise
352     x = torch.randn(n_samples, self.n_ctrl, 2, device=device)
353     # MMD improvement Q: initialise velocity as constant-velocity
baseline (startgoal)
354     _t = torch.linspace(0, 1, self.n_ctrl, device=device).view(1, -1,
355     1)
356     self._velocity_buf = ((goal_rep.unsqueeze(1) -
357     start_rep.unsqueeze(1))
358     * _t).detach()    # (n_samples, n_ctrl, 2)
359
360     # DDIM schedule: T 0 with ddim_steps
361     ddim_steps = cfg.ddim_steps
362     step_size  = max(1, sched.T // ddim_steps)
363     timesteps  = list(range(sched.T - 1, -1, -step_size))
364
365     for i, t_val in enumerate(timesteps):
366         t_batch = torch.full((n_samples,), t_val, device=device,
367         dtype=torch.long)
368         t_prev  = timesteps[i + 1] if i + 1 < len(timesteps) else -1
369
370         eps = self.score_net(x, t_batch, cond)
371
372         # Cost guidance (MPD-splines + MMD)
373         if use_guidance:

```



```

371         with torch.enable_grad():
372             x_g = x.detach().requires_grad_(True)
373             eps_g = self.score_net(x_g, t_batch, cond)
374             x0_est = sched.predict_x0_from_eps(x_g, t_batch,
eps_g)
375             traj = self.bspline(x0_est)          # (n_samples,
n_eval, 2)
376
377             # Smoothness cost
378             s_cost = smoothness_loss(traj)
379             s_grad = torch.autograd.grad(s_cost, x_g,
retain_graph=True)[0]
380             # MMD improvement P: zero gradient at start/goal
endpoints
381             s_grad[..., 0, :] = 0.0
382             s_grad[..., -1, :] = 0.0
383
384             # MMD improvement N: L1 goal-reach guidance
(GuideTrajectoryLastPoint)
385             # objective = -|goal - traj[-1]|.sum(-1) more robust
than L2
386             goal_r = goal_rep if goal_rep.shape[0] ==
traj.shape[0]          else
goal_rep.expand(traj.shape[0], -1)
387             l1_cost = torch.abs(goal_r - traj[:, -1,
:]).sum(-1).mean()
388             l1_grad = torch.autograd.grad(l1_cost, x_g,
retain_graph=True)[0]
389             l1_grad[..., 0, :] = 0.0    # pin start
390             l1_grad[..., -1, :] = 0.0    # pin goal (it's the
target - no push needed)
391
392             # MPD-splines improvement O: acceleration
minimisation guidance

```

```

393         # GuideSmoothnessFiniteDifferenceVelocity: acc =
diff(vel), minimise ||acc||
394         vel = traj[:, 1:, :] - traj[:, :-1, :]          #
(n, n_eval-1, 2)
395         acc = vel[:, 1:, :] - vel[:, :-1, :]          #
(n, n_eval-2, 2)
396         acc_cost = torch.linalg.norm(acc,
dim=-1).sum(-1).mean()
397         acc_grad = torch.autograd.grad(acc_cost, x_g)[0]
398         acc_grad[..., 0, :] = 0.0
399         acc_grad[..., -1, :] = 0.0
400
401         # MMD improvement R: eCBS soft constraint - other
agents' paths
402         # Soft weighting: exp(-temporal_gap * 0.1) so
near-future conflicts
403         # are penalised more strongly than far-future ones
404         if other_trajs is not None:
405             traj_np = traj.detach()
406             ot = other_trajs
407             if ot.dim() == 2:
408                 ot = ot.unsqueeze(1).expand(-1,
traj_np.shape[1], -1)
409             all_t = torch.cat([traj_np, ot], dim=0)
410             c_grad_all = inter_robot_collision_gradient(
411                 all_t, radius=cfg.robot_collision_radius)
412             c_grad = c_grad_all[:n_samples]
413             # eCBS soft weight: closer timesteps get stronger
penalty
414             n_eval = traj_np.shape[1]
415             t_idx = torch.arange(n_eval, device=device,
dtype=torch.float32)
416             # Exponential soft weight: near future matters
most

```

```

417         soft_w = torch.exp(-t_idx * 0.05).view(1, n_eval,
1)
418         c_grad = c_grad * soft_w
419         B_mat = self.bspline._get_basis(device)
420         c_grad_ctrl =
torch.matmul(B_mat.t().unsqueeze(0), c_grad)
421     else:
422         c_grad_ctrl = torch.zeros_like(x_g)
423
424         # MMD improvement Q: update velocity estimate (gradient
descent on vel)
425         if self._velocity_buf is not None:
426             vel_grad = torch.diff(s_grad, dim=-2)
# (n, n_ctrl-1, 2)
427             vel_grad_pad = torch.cat(
428                 [vel_grad, torch.zeros_like(vel_grad[..., :1,
: ])], dim=-2)
429             self._velocity_buf = (self._velocity_buf - 0.01 *
vel_grad_pad).detach()
430
431             eps = (eps
432                   + cfg.guidance_weight_smoothness * s_grad
433                   + cfg.guidance_weight_collision * c_grad_ctrl
434                   + cfg.guidance_weight_goal_l1 * l1_grad
435                   + cfg.guidance_weight_accel * acc_grad)
436
437             # DDIM step
438             x = sched.ddim_step(x, t_val, t_prev, eps)
439
440             # Evaluate B-spline to get waypoints
441             traj = self.bspline(x) # (n_samples, n_eval, 2)
442             return traj
443
444 def _infer_traj(self, obs: torch.Tensor,

```

```

445         start_pos: torch.Tensor,
446         goal_pos: torch.Tensor) -> torch.Tensor:
447     """
448     TorchScript-compatible no-guidance DDIM inference.
449
450     This is the export/runtime path. It deliberately does NOT do cost
451     guidance - guidance requires torch.autograd.grad which is not
452     scriptable. Instead we run a pure-tensor DDIM loop using the
453     _sc_acp buffer registered in __init__.
454
455     Inputs are shaped (1, D) - single-example inference only; the
456     simulation side (SimReadyDiff) always passes batch=1.
457
458     Args:
459     obs: (1, obs_dim)
460     start_pos: (1, 2)
461     goal_pos: (1, 2)
462     Returns:
463     traj: (1, n_eval, 2)
464     """
465     device = obs.device
466     n_samples = 1
467     cond = self._make_cond(obs, start_pos, goal_pos) # (1, cond_dim)
468
469     # Start from pure Gaussian noise
470     x = torch.randn(n_samples, self.n_ctrl, 2, device=device)
471
472     # DDIM integer schedule: step_size = T // ddim_steps, at least 1.
473     # Equivalent to: timesteps = list(range(T-1, -1, -step_size))
474     T = self._T_steps
475     ddim = self._ddim_steps
476     step_size = T // ddim
477     if step_size < 1:
478         step_size = 1

```

```

479         n_steps = (T - 1) // step_size + 1
480
481         acp = self._sc_acp    # (T,) float32 buffer
482
483         for i in range(n_steps):
484             t        = (T - 1) - i * step_size
485             # t_prev follows the original range(..., -1, -step_size): the
next
486             # entry in the descending list, or -1 if we've consumed the
last.
487             if i + 1 < n_steps:
488                 t_prev = (T - 1) - (i + 1) * step_size
489             else:
490                 t_prev = -1
491
492             t_batch = torch.full((n_samples,), t, device=device,
dtype=torch.long)
493             eps = self.score_net(x, t_batch, cond)
494
495             # Inline DDIM deterministic step (was
DiffusionSchedule.ddim_step)
496             acp_t = acp[t]
497             if t_prev >= 0:
498                 acp_prev = acp[t_prev]
499             else:
500                 acp_prev = torch.ones((), device=device, dtype=acp.dtype)
501
502             sqrt_acp_t        = torch.sqrt(acp_t)
503             sqrt_one_m_acp_t = torch.sqrt(1.0 - acp_t)
504             pred_x0 = (x - sqrt_one_m_acp_t * eps) / sqrt_acp_t
505             pred_x0 = torch.clamp(pred_x0, -1.0, 1.0)
506
507             x = torch.sqrt(acp_prev) * pred_x0 + torch.sqrt(1.0 -
acp_prev) * eps

```

```

508
509     # Evaluate spline
510     traj = self.bspline(x)                # (1, n_eval, 2)
511     return traj
512
513     def decode_task_id(self, traj: torch.Tensor,
514                       task_locs: torch.Tensor,
515                       other_robot_locs: Optional[torch.Tensor] = None)
516     -> Tuple[torch.Tensor, torch.Tensor]:
517         """
518         Decode the nearest task to the trajectory endpoint.
519         Returns (robot_id=0, task_id_local).
520
521         Args:
522         traj: (n_eval, 2) - normalised trajectory
523         task_locs: (N, 2) - normalised active task locations
524         other_robot_locs: unused, kept for API compatibility
525
526         Returns:
527         (robot_id, task_id): both scalar tensors
528         """
529         endpoint = traj[-1]                # (2,)
530         dists    = torch.norm(task_locs - endpoint, dim=-1)  # (N,)
531         task_id  = torch.argmax(dists)
532         robot_id = torch.zeros(1, dtype=torch.long,
533                               device=traj.device).squeeze()
534         return robot_id, task_id
535
536     # Concrete models
537
538     class TDM(DiffusionBase):
539         """
540         Trajectory Diffusion Model.

```

```

540 Vanilla DDPM + cosine schedule + B-spline + goal conditioning.
541 """
542     name = 'TDM'
543
544     def __init__(self, **kwargs):
545         kwargs.setdefault('schedule', 'cosine')
546         super().__init__('TDM', **kwargs)
547
548     def forward(self, obs: torch.Tensor,
549                 task_locs: torch.Tensor,
550                 other_robot_locs: torch.Tensor) -> Tuple[torch.Tensor,
551                 torch.Tensor]:
552         """
553         TorchScript-compatible inference.
554         Returns (robot_id=0, task_id_local).
555         """
556         if task_locs.shape[0] == 0:
557             return torch.zeros(1, dtype=torch.long).squeeze(),\
558                    torch.zeros(1, dtype=torch.long).squeeze()
559
560         # Use first task as goal for trajectory generation
561         goal_pos = task_locs[0:1]          # (1, 2)
562         start_pos = torch.zeros(1, 2, device=obs.device)
563
564         # Scriptable DDIM path (no guidance; guidance uses autograd.grad
565         # which cannot be scripted). For unguided inference at export time
566         # this is equivalent to the old sample(..., use_guidance=False)
567         call.
568         # obs arrives already 2D (SimReadyDiff batches it); do NOT
569         unsqueeze
570         # again or ObservationEncoder._parse_global_obs reshape will
571         crash.
572         traj = self._infer_traj(obs, start_pos, goal_pos) # (1, n_eval,
573         2)

```

```

569         return self.decode_task_id(traj[0], task_locs)
570
571
572 class DiffusionPolicyModel(DiffusionBase):
573     """
574     DiffusionPolicy - action-conditioned diffusion.
575     no longer uses 1001-dim one-hot.
576     Now uses B-spline control points output = (n_ctrl × 2) = 16 dims.
577     Conditioned on global obs + all task locations.
578     Uses linear noise schedule (vs TDM cosine) so training dynamics and
579     HybridDiffusion distillation pull from a genuinely different source.
580     """
581     name = 'DiffusionPolicy'
582
583     def __init__(self, **kwargs):
584         kwargs.setdefault('schedule', 'linear')
585         super().__init__('DiffusionPolicy', **kwargs)
586         # Additional task attention for direct task scoring from
trajectory
587         self.task_head = nn.Linear(cfg.embed_dim, 1)
588
589     def forward(self, obs: torch.Tensor,
590                task_locs: torch.Tensor,
591                other_robot_locs: torch.Tensor) -> Tuple[torch.Tensor,
592                torch.Tensor]:
593         if task_locs.shape[0] == 0:
594             return torch.zeros(1, dtype=torch.long).squeeze(), \
595                    torch.zeros(1, dtype=torch.long).squeeze()
596
597         # Cap at 5 candidate tasks (TorchScript-safe int math, no min over
# a Python int + tensor.shape mix which TS sometimes rejects).
598         n_check: int = task_locs.shape[0]
599         if n_check > 5:
600             n_check = 5

```



```

601
602     best_tid_int: int = 0
603     # device-aware scalar; the old torch.tensor(-inf) was a CPU
scalar and
604     # would mismatch obs.device on a GPU export.
605     best_score = torch.full((), -float('inf'), device=obs.device)
606     start = torch.zeros(1, 2, device=obs.device)
607     # obs arrives already 2D (1, D) from SimReadyDiff; no unsqueeze
needed.
608
609     for tid in range(n_check):
610         goal = task_locs[tid:tid+1]
611         # Scriptable no-guidance DDIM (replaces self.sample(...,
use_guidance=False))
612         traj = self._infer_traj(obs, start, goal)
613         # Score: how close endpoint is to goal
614         score = -torch.norm(traj[0, -1] - goal.squeeze(0))
615         if score > best_score:
616             best_score = score
617             best_tid_int = tid
618
619     best_tid = torch.tensor(best_tid_int, dtype=torch.long)
620     return torch.zeros(1, dtype=torch.long).squeeze(), best_tid
621
622
623 class Diffuser(DiffusionBase):
624     """
625     Diffuser - state-conditioned sequence diffusion.
626     Generates full (state, action) trajectory conditioned on initial obs.
627     Uses cosine schedule + deeper ScoreNet (hidden_dim 256→384) to differ
628     from TDM (same schedule but shallower) for HybridDiffusion diversity.
629     """
630     name = 'Diffuser'
631

```

```

632     def __init__(self, **kwargs):
633         kwargs.setdefault('schedule', 'cosine')
634         kwargs.setdefault('hidden_dim', 384)
635         super().__init__('Diffuser', **kwargs)
636
637     def forward(self, obs: torch.Tensor,
638                 task_locs: torch.Tensor,
639                 other_robot_locs: torch.Tensor) -> Tuple[torch.Tensor,
640                 torch.Tensor]:
641         if task_locs.shape[0] == 0:
642             return torch.zeros(1, dtype=torch.long).squeeze(), \
643                    torch.zeros(1, dtype=torch.long).squeeze()
644
645         goal = task_locs[0:1]
646         start = torch.zeros(1, 2, device=obs.device)
647         # Scriptable DDIM (no guidance; autograd.grad is not
648         TorchScript-safe).
649         # obs is already 2D from SimReadyDiff; don't unsqueeze.
650         traj = self._infer_traj(obs, start, goal)
651         return self.decode_task_id(traj[0], task_locs)
652
653 class SDE(DiffusionBase):
654     """
655     Score-based SDE diffusion using Variance Exploding (VE) SDE formulation.
656      $\sigma(t)$  schedule instead of  $\alpha$ , providing continuous-time diffusion.
657     """
658     name = 'SDE'
659
660     def __init__(self, sigma_min: float = 0.002, sigma_max: float = 3.0,
661                 **kwargs):
662         super().__init__('SDE', **kwargs)
663         self.sigma_min = sigma_min
664         self.sigma_max = sigma_max

```

```

663
664 @torch.jit.unused
665 def sigma(self, t: torch.Tensor) -> torch.Tensor:
666     """VE SDE:  $\sigma(t) = \sigma_{\min} * (\sigma_{\max}/\sigma_{\min})^t$ """
667     return self.sigma_min * (self.sigma_max / self.sigma_min) ** t
668
669 @torch.jit.unused
670 def training_loss(self, traj_wpts, obs, start_pos, goal_pos):
671     """VE-SDE training with  $\epsilon$ -prediction for numerical stability.
672
673     Uses VE noise (larger dynamic range:  $\sigma$  up to 50) but predicts
674     normalised noise so targets are always  $\sim N(0, I)$ 
675     scale ( $\sim 1$ ), matching the base-class DDPM _infer_traj path.
676
677     The original score-matching target reaches scale  $\sim 100$  at
678      $\sigma_{\min}=0.01$ , making loss  $\sim 1277$  even though the weight should cancel
679     in theory --- in practice the ScoreNet's residual activations
grow with
680     large- $\sigma$  inputs before LayerNorm kicks in, breaking the
cancellation.
681
682     """
683     sched = self._get_schedule(obs.device)
684     batch = obs.shape[0]
685     device = obs.device
686     bspline = self.bspline
687     B_mat = bspline._get_basis(device)
688     BtB = B_mat.t().mm(B_mat)
689     Bt = B_mat.t()
690     _orig_dtype = traj_wpts.dtype
691     ctrl_pts = torch.linalg.solve(
692         BtB.unsqueeze(0).expand(batch, -1, -1).float(),
693         torch.matmul(Bt.unsqueeze(0).expand(batch, -1, -1).float(),
694             traj_wpts.float()).float(),
695     ).to(_orig_dtype)

```

```

695
696     t_cont = torch.rand(batch, device=device)
697     sigma_t = self.sigma(t_cont).unsqueeze(-1).unsqueeze(-1) #
        (batch,1,1)
698     eps     = torch.randn_like(ctrl_pts)                        #  $\epsilon \sim$ 
         $N(0,I)$ , scale 1
699     x_noisy = ctrl_pts + sigma_t * eps                          # VE
        forward process
700
701     t_int   = (t_cont * (sched.T - 1)).long()
702     cond    = self._make_cond(obs, start_pos, goal_pos)
703     eps_hat = self.score_net(x_noisy, t_int, cond)
704
705     # Predict normalised noise  $\epsilon$  (scale  $\sim 1$ ); loss stabilises to  $\sim 1$ 
        like other models.
706     mse_loss = F.mse_loss(eps_hat, eps)
707
708     #  $x_0$  estimate for smoothness:
709     x0_est   = (x_noisy - sigma_t * eps_hat).clamp(-5.0, 5.0)
710     traj_est = bspline(x0_est)
711     smooth_l = smoothness_loss(traj_est)
712     return mse_loss + cfg.smoothness_loss_weight * smooth_l
713
714 def forward(self, obs: torch.Tensor,
715             task_locs: torch.Tensor,
716             other_robot_locs: torch.Tensor) -> Tuple[torch.Tensor,
        torch.Tensor]:
717     if task_locs.shape[0] == 0:
718         return torch.zeros(1, dtype=torch.long).squeeze(),\
719             torch.zeros(1, dtype=torch.long).squeeze()
720     goal = task_locs[0:1]
721     start = torch.zeros(1, 2, device=obs.device)
722     # Export-time DDIM (scriptable). NOTE: this reuses the base DDPM
723     #  $\alpha$ -cumprod buffer from DiffusionBase, not the VE  $\sigma(t)$  schedule -

```

```

724     # torch.autograd.grad/integrated-sigma paths are not scriptable.
725     # SDE's training-time speciality is preserved
726     # in training_loss; the exported model is a well-trained score
727     # network unrolled under a DDPM schedule, which is close in
728     # practice and matches how DiffusionPolicy exports.
729     # obs is already 2D (1, D) from SimReadyDiff; do not unsqueeze.
730     traj = self._infer_traj(obs, start, goal)
731     return self.decode_task_id(traj[0], task_locs)
732
733
734 class HybridDiffusion(nn.Module):
735     """
736     HybridDiffusion: ensemble of ALL 4 base diffusion models
737     (TDM, DiffusionPolicy, Diffuser, SDE) with adaptive model selection.
738
739     At inference:
740     - All 4 models generate trajectory candidates
741     - Each is scored on: endpoint proximity to task + smoothness + collision
742       cost
743     - Best trajectory wins
744
745     The ensemble captures multiple trajectory distribution modes, making it
746     more robust than any single diffusion model.
747     """
748
749     name = 'HybridDiffusion'
750
751     def __init__(self, obs_dim: int = GLOBAL_OBS_DIM, **kwargs):
752         super().__init__()
753         self.tdm = TDM(obs_dim=obs_dim, **kwargs)
754         self.dp = DiffusionPolicyModel(obs_dim=obs_dim, **kwargs)
755         self.diff = Diffuser(obs_dim=obs_dim, **kwargs)
756         self.sde = SDE(obs_dim=obs_dim, **kwargs)
757
758         # Adaptive model weights - updated during training based on loss

```

```

757         self.register_buffer('model_weights', torch.ones(4) / 4.0)
758
759     @torch.jit.unused
760     def _score_traj(self, traj: torch.Tensor,
761                     goal: torch.Tensor,
762                     smooth_weight: float = 1.0) -> float:
763         """
764         Score a trajectory: higher = better.
765         -dist_to_goal - smooth_weight * smoothness
766         """
767         endpoint = traj[-1] if traj.dim() == 2 else traj[0, -1]
768         dist = float(torch.norm(endpoint - goal.squeeze()))
769         sm = float(smoothness_loss(traj.unsqueeze(0) if traj.dim() == 2
770                                   else traj))
771         return -(dist + smooth_weight * sm)
772
773     @torch.jit.unused
774     def training_loss(self, traj_wpts, obs, start_pos, goal_pos):
775         """
776         Weighted average loss from all 4 members.
777         Models that performed better (lower loss) get higher weight next time.
778         """
779         losses = torch.stack([
780             self.tdm.training_loss(traj_wpts, obs, start_pos, goal_pos),
781             self.dp.training_loss(traj_wpts, obs, start_pos, goal_pos),
782             self.diff.training_loss(traj_wpts, obs, start_pos, goal_pos),
783             self.sde.training_loss(traj_wpts, obs, start_pos, goal_pos),
784         ])
785
786         # Update adaptive weights: models with lower loss get higher
787         weight
788         with torch.no_grad():
789             inv_loss = 1.0 / (losses.detach() + 1e-6)
790             self.model_weights = (inv_loss / inv_loss.sum()).clamp(0.05,

```

```

0.6)

789
790     # Weighted average loss
791     return (self.model_weights.detach() * losses).sum()
792
793     def forward(self, obs: torch.Tensor,
794                 task_locs: torch.Tensor,
795                 other_robot_locs: torch.Tensor) -> Tuple[torch.Tensor,
796                 torch.Tensor]:
797         """
798         Ensemble inference (TorchScript-compatible): each of the 4 base
799         diffusion submodels generates one trajectory via the no-guidance
800         DDIM path (_infer_traj), scored by weighted (endpoint distance +
801         smoothness), and the best trajectory wins.
802
803         Rewritten from the 'for model, w in zip(members, weights)' loop -
804         TorchScript cannot iterate a Python list of heterogeneous nn.Modules.
805         The 'other_trajs' collision-guidance arg is also dropped here because
806         cost guidance itself is not scriptable (torch.autograd.grad inside
807         sample); this matches how the single-model forwards handle it.
808         """
809         if task_locs.shape[0] == 0:
810             return torch.zeros(1, dtype=torch.long).squeeze(),\
811                    torch.zeros(1, dtype=torch.long).squeeze()
812
813         goal = task_locs[0:1]                                # (1, 2)
814         start = torch.zeros(1, 2, device=obs.device)
815         # obs arrives already 2D (1, D) from SimReadyDiff; do not
816         unsqueeze.
817         goal_v = goal.squeeze(0)                              # (2,)
818
819         # 4 candidate trajectories (each shape (1, n_eval, 2))
820         t0 = self.tdm._infer_traj(obs, start, goal)
821         t1 = self.dp._infer_traj(obs, start, goal)

```

```

820     t2 = self.diff._infer_traj(obs, start, goal)
821     t3 = self.sde._infer_traj(obs, start, goal)
822
823     w = self.model_weights                # (4,)
824
825     # Inline per-candidate score:
826     # (was self._score_traj; inlined because that helper uses float
827     # coercion and.dim branching which TorchScript rejects.)
828     ep0 = t0[0, -1]
829     d0 = torch.norm(ep0 - goal_v)
830     d2_0 = t0[:, 2:, :] - 2.0 * t0[:, 1:-1, :] + t0[:, :-2, :]
831     sm0 = (d2_0 ** 2).mean()
832     s0 = -(d0 + sm0) * w[0]
833
834     ep1 = t1[0, -1]
835     d1 = torch.norm(ep1 - goal_v)
836     d2_1 = t1[:, 2:, :] - 2.0 * t1[:, 1:-1, :] + t1[:, :-2, :]
837     sm1 = (d2_1 ** 2).mean()
838     s1 = -(d1 + sm1) * w[1]
839
840     ep2 = t2[0, -1]
841     d2_ = torch.norm(ep2 - goal_v)
842     d2_2 = t2[:, 2:, :] - 2.0 * t2[:, 1:-1, :] + t2[:, :-2, :]
843     sm2 = (d2_2 ** 2).mean()
844     s2 = -(d2_ + sm2) * w[2]
845
846     ep3 = t3[0, -1]
847     d3 = torch.norm(ep3 - goal_v)
848     d2_3 = t3[:, 2:, :] - 2.0 * t3[:, 1:-1, :] + t3[:, :-2, :]
849     sm3 = (d2_3 ** 2).mean()
850     s3 = -(d3 + sm3) * w[3]
851
852     # Pick argmax score
853     best_traj = t0

```



```

854         best_score = s0
855         if s1 > best_score:
856             best_score = s1
857             best_traj = t1
858         if s2 > best_score:
859             best_score = s2
860             best_traj = t2
861         if s3 > best_score:
862             best_score = s3
863             best_traj = t3
864
865         return self.tdm.decode_task_id(best_traj[0], task_locs)
866
867
868     # Factory
869
870     def build_diffusion_model(name: str, **kwargs) -> nn.Module:
871         models = {
872             'TDM': TDM,
873             'DiffusionPolicy': DiffusionPolicyModel,
874             'Diffuser': Diffuser,
875             'SDE': SDE,
876             'HybridDiffusion': HybridDiffusion,
877         }
878         if name not in models:
879             raise ValueError(f'Unknown diffusion model: {name}')
880         return models[name](**kwargs)

```

Listing B.14: All five diffusion model classes: TDM, DiffusionPolicy, Diffuser, SDE and HybridDiffusion with shared score network and DDIM sampler.

```

1  """
2  models/sim_ready_wrappers.py

```

```

3 =====
4 TorchScript-exportable wrappers that produce .pt files for deployment.
5
6 The sim code will be written later and will load these files; the wrapper
7 design targets TorchScript compatibility as a technical requirement.
8
9     Update
10
11 these wrappers as the model architecture evolves --- the sim side will be
12 updated to match when it is built.
13
14 export_sim_ready() accepts either rl OR diff as None so the caller
15     (run.py)
16
17 can export just RL.pt or just diff.pt files in a single call:
18
19     * every (algo, scenario) gets an RL export -> 15 sim_*.pt files
20     * every (diff, HybridRL, scenario) gets one -> 15 sim_diff_*.pt files
21     Total 30 sim-ready files.
22
23 torch.jit.script is used directly (no trace fallback). If scripting fails
24 that is a model-authoring error (unsupported op, type ambiguity) and must
25 surface immediately so it can be fixed before deployment.
26
27 Naming:
28
29     RL:          sim_{ALGO}_{scenario}.pt
30     Diffusion:   sim_diff_{DIFF}_{RL}_{scenario}.pt
31
32 """
33
34 from __future__ import annotations
35
36 import os
37
38 from typing import Final, Tuple, Optional, Dict
39
40
41 import torch
42
43 import torch.nn as nn
44
45
46 from config.base_config import (
47     MAX_ROBOTS, MAX_TASKS, GLOBAL_OBS_DIM, LOCAL_OBS_DIM, FLAT_OBS_DIM,
48     SCENARIO_NORM, SIM_READY_DIR,

```

```

35 )
36
37
38 class SimReadyRL(nn.Module):
39     """
40     TorchScript-exportable RL wrapper. Embeds scenario-specific normalisation
41     and dispatches to the right input dim per algorithm.
42
43     forward(raw_obs) -> (robot_id, task_id)
44     """
45
46     def __init__(self, model: nn.Module, algo: str, scenario: str):
47         super().__init__()
48         self.model = model
49         self.algo = algo
50         self.scenario = scenario
51         norm = SCENARIO_NORM[scenario]
52         self.register_buffer('grid_half',
53                               torch.tensor(norm['grid_half']))
54         self.register_buffer('max_speed',
55                               torch.tensor(norm['max_speed']))
56         self.register_buffer('battery_cap',
57                               torch.tensor(norm['battery_capacity']))
58         self.register_buffer('max_payload',
59                               torch.tensor(norm['max_payload']))
60
61         # TorchScript requires int constants as instance attrs, not
62         module globals
63
64         self.local_obs_dim: int = LOCAL_OBS_DIM
65         self.flat_obs_dim: int = FLAT_OBS_DIM
66
67     def forward(self, obs: torch.Tensor,
68                 mask: Optional[torch.Tensor] = None) ->
69         Tuple[torch.Tensor, torch.Tensor]:
70
71         obs = obs.float()

```

```

63         if obs.dim() == 1:
64             obs = obs.unsqueeze(0)    # (1, D)
65
66             # observation_extractors already divides raw positions by
grid_half;
67             # this is a belt-and-braces soft normalisation for any raw ROS obs
# that reach this wrapper un-normalised.
68
69             obs_norm = torch.clamp(obs / (self.grid_half + 1e-6), -10.0, 10.0)
70             D = obs_norm.shape[-1]
71
72             # Always compute both input views; each model forward ignores
whichever
73             # it does not need. Uniform two-arg call avoids TorchScript's
74             # whole-function type-checking rejecting the second-arg branch
when the
75             # stored model has a single-arg forward (e.g. MAPPO compiled with
a
76             # HybridRL branch visible).
77             local_in = (obs_norm[:, :self.local_obs_dim] if D >=
self.local_obs_dim
78                     else torch.cat(
79                         [obs_norm,
80                         obs_norm.new_zeros(1, self.local_obs_dim - D)],
dim=-1))
81             flat_in = (obs_norm[:, :self.flat_obs_dim] if D >=
self.flat_obs_dim
82                     else torch.cat(
83                         [obs_norm,
84                         obs_norm.new_zeros(1, self.flat_obs_dim - D)],
dim=-1))
85
86             if self.algo in ('MAPPO', 'QMIX'):
87                 robot_id, task_id = self.model(local_in, flat_in, mask)
88             elif self.algo in ('DQN', 'MADDPG'):

```

```

89         robot_id, task_id = self.model(flat_in, local_in, mask)
90     elif self.algo == 'HybridRL':
91         robot_id, task_id = self.model(local_in, flat_in, mask)
92     else:
93         raise ValueError('SimReadyRL: unknown algo ' + str(self.algo))
94
95     return robot_id.squeeze().long(), task_id.squeeze().long()
96
97
98 class SimReadyDiff(nn.Module):
99     """
100     TorchScript-exportable diffusion wrapper.
101
102     forward(obs, task_locs, other_robot_locs) -> (robot_id=0, task_id_local)
103     """
104
105     def __init__(self, model: nn.Module, diff_algo: str, rl_algo: str,
106                 scenario: str):
107         super().__init__()
108         self.model = model
109         self.diff_algo = diff_algo
110         self.rl_algo = rl_algo
111         self.scenario = scenario
112         self.global_obs_dim: int = GLOBAL_OBS_DIM
113
114     def forward(self, obs: torch.Tensor,
115                task_locs: torch.Tensor,
116                other_robot_locs: torch.Tensor
117                ) -> Tuple[torch.Tensor, torch.Tensor]:
118         obs = obs.float()
119         tloc = task_locs.float()
120         oloc = other_robot_locs.float()
121
122         if obs.dim() == 1:

```

```

123         obs_g = obs[:self.global_obs_dim].unsqueeze(0)
124     else:
125         obs_g = obs[:, :self.global_obs_dim]
126
127     # Guard: clamp task/robot positions to [-1.5, 1.5] normalised
128     space.
129     tloc = torch.clamp(tloc, -1.5, 1.5)
130     oloc = torch.clamp(oloc, -1.5, 1.5)
131
132     # Ensure at least one task and one other robot (avoid empty
133     tensor paths)
134     if tloc.numel() == 0:
135         tloc = torch.zeros(1, 2)
136     if oloc.numel() == 0:
137         oloc = torch.zeros(1, 2)
138
139     robot_id, task_id = self.model(obs_g, tloc, oloc)
140     return robot_id.long(), task_id.long()
141
142 #
143 # Public export API - keyword-driven, either half can be None
144 #
145 def export_sim_ready(rl_model: Optional[nn.Module],
146                     diff_model: Optional[nn.Module],
147                     algo: Optional[str],
148                     diff_algo: Optional[str],
149                     scenario: str,
150                     output_dir: str = SIM_READY_DIR,
151                     paired_rl: Optional[str] = None) -> Dict[str, str]:
152     """
153     Export RL and/or diffusion models as TorchScript.pt files.
154 
```

```

155 semantics:
156 - To export only an RL model:
157 export_sim_ready(rl, None, algo, None, scenario)
158 - To export only a diffusion model:
159 export_sim_ready(None, dm, None, diff_algo, scenario,
160 paired_rl=DIFF_RL_PAIR)
161 - To export both in one call (legacy behaviour):
162 export_sim_ready(rl, dm, algo, diff_algo, scenario)
163
164 Returns a dict with 'rl' and/or 'diff' keys mapping to saved paths.
165 At least one of (rl_model, diff_model) must be non-None, and the
166 corresponding algo name must be provided.
167 """
168     if rl_model is None and diff_model is None:
169         raise ValueError('export_sim_ready: must pass at least one of '
170                            '(rl_model, diff_model)')
171
172     os.makedirs(output_dir, exist_ok=True)
173     out: Dict[str, str] = {}
174
175     # RL export
176     if rl_model is not None:
177         if algo is None:
178             raise ValueError('export_sim_ready: rl_model given but algo
179                                is None')
180
181         rl_wrapper = SimReadyRL(rl_model, algo, scenario)
182         rl_wrapper.eval()
183
184         # no silent try/except around scripting. If script fails, that's
185         # a model-authoring error and must surface.
186         rl_scripted = torch.jit.script(rl_wrapper)
187         rl_filename = f'sim_{algo}_{scenario}.pt'
188         rl_path = os.path.join(output_dir, rl_filename)
189         rl_scripted.save(rl_path)
190         out['rl'] = rl_path

```

```

188
189     # Diffusion export
190     if diff_model is not None:
191         if diff_algo is None:
192             raise ValueError('export_sim_ready: diff_model given but '
193                               'diff_algo is None')
194         # The RL name baked into the filename: prefer paired_rl (v4
195         diff-only
196         # path), fall back to algo (v3 combined-call path).
197         rl_name = paired_rl if paired_rl is not None else algo
198         if rl_name is None:
199             raise ValueError('export_sim_ready: diff_model given but
200                               'neither '
201                               'paired_rl nor algo was provided to name the
202                               file')
203         diff_wrapper = SimReadyDiff(diff_model, diff_algo, rl_name,
204                                     scenario)
205         diff_wrapper.eval()
206         diff_scripted = torch.jit.script(diff_wrapper)
207         diff_filename = f'sim_diff_{diff_algo}_{rl_name}_{scenario}.pt'
208         diff_path = os.path.join(output_dir, diff_filename)
209         diff_scripted.save(diff_path)
210         out['diff'] = diff_path
211
212     return out

```

Listing B.15: TorchScript wrappers that produce simulation-ready checkpoints from the trained PyTorch models.

Package: training

```

1 # training/__init__.py

```

Listing B.16: Package init for the training package.

```
1  """
2  training/rl_trainer.py
3  =====
4  Shared RL training loop for all five agents across all three scenarios.
5
6  Key design choices:
7      - Invalid action masking applied before every policy update (TA-RWARE).
8      - Greedy REINFORCE baseline reduces variance in sparse-reward scenarios
9        (DCMRTA).
10     - Imitation learning mixing decays exponentially from il_mix_rate to
11       il_min_rate; the floor prevents policy collapse when reward signal is
12       weak.
13     - GAE advantage estimation with clip_epsilon PPO clipped surrogate.
14     - CUDA memory fraction set via AQUA_GPU_GB_PER_WORKER env var.
15  """
16  import logging
17  import os
18  import time
19  import torch
20  import torch.nn as nn
21  import numpy as np
22  from pathlib import Path
23  from typing import Dict, Any, Optional, List, Tuple
24
25  logger = logging.getLogger('aqua.rl_trainer')
26
27  from config.base_config import (
28      RLConfig, SCENARIO_CONFIGS, MAX_ROBOTS, MAX_TASKS, ACTION_DIM,
29      LOCAL_OBS_DIM, GLOBAL_OBS_DIM, FLAT_OBS_DIM,
30      RESULTS_DIR, MODELS_DIR, IL_DECAY_EXPONENT,
31      ROBOT_STATUS_FAILED, TASK_STATUS_COMPLETE,
```

```

31 )
32 from utils.action_masking import ActionMaskWrapper
33 from utils.reproducibility import ReproducibilityManager
34 from environment.reward_shaping import RewardShapingWrapper
35
36 cfg = RLConfig()
37
38
39 class ReplayBuffer:
40     """Circular replay buffer for DQN/MADDPG with TD bootstrapping."""
41
42     def __init__(self, capacity: int = 50_000):
43         self.capacity = capacity
44         self._obs: List = []
45         self._next_obs: List = []
46         self._actions: List = []
47         self._rewards: List = []
48         self._dones: List = []
49         self._ptr = 0
50
51     def add(self, obs, action, reward, next_obs, done):
52         if len(self._obs) < self.capacity:
53             self._obs.append(obs)
54             self._next_obs.append(next_obs)
55             self._actions.append(action)
56             self._rewards.append(reward)
57             self._dones.append(done)
58         else:
59             p = self._ptr % self.capacity
60             self._obs[p] = obs
61             self._next_obs[p] = next_obs
62             self._actions[p] = action
63             self._rewards[p] = reward
64             self._dones[p] = done

```

```

65         self._ptr += 1
66
67     def sample(self, batch_size: int):
68         idx = np.random.choice(len(self._obs), size=batch_size,
69                                replace=False)
69         return (
70             np.array([self._obs[i]      for i in idx]),
71             np.array([self._actions[i]   for i in idx]),
72             np.array([self._rewards[i]   for i in idx], dtype=np.float32),
73             np.array([self._next_obs[i]  for i in idx]),
74             np.array([self._dones[i]     for i in idx], dtype=np.float32),
75         )
76
77     def __len__(self):
78         return len(self._obs)
79
80
81 class RolloutBuffer:
82     """Stores one episode of experience for PPO/REINFORCE update."""
83
84     def __init__(self):
85         self.local_obs: List = []
86         self.global_obs: List = []
87         self.flat_obs: List = []
88         self.actions: List = []
89         self.log_probs: List = []
90         self.rewards: List = []
91         self.values: List = []
92         self.dones: List = []
93         self.masks: List = []
94
95     def add(self, local_obs, global_obs, flat_obs, action, log_prob,
96            reward, value, done, mask=None):
97         self.local_obs.append(local_obs)

```

```

98     self.global_obs.append(global_obs)
99     self.flat_obs.append(flat_obs)
100    self.actions.append(action)
101    self.log_probs.append(log_prob)
102    self.rewards.append(reward)
103    self.values.append(value)
104    self.dones.append(done)
105    self.masks.append(mask)
106
107    def compute_returns(self, gamma: float = cfg.gamma,
108                       lam: float = cfg.gae_lambda,
109                       last_value: float = 0.0) -> Tuple[torch.Tensor,
110               torch.Tensor]:
111        """Compute GAE advantages and discounted returns."""
112        rewards = self.rewards
113        values = self.values + [last_value]
114        dones = self.dones
115
116        advantages = []
117        gae = 0.0
118        for t in reversed(range(len(rewards))):
119            delta = (rewards[t] + gamma * values[t + 1] * (1 - dones[t])
120                - values[t])
121            gae = delta + gamma * lam * (1 - dones[t]) * gae
122            advantages.insert(0, gae)
123
124        returns = [adv + val for adv, val in zip(advantages,
125            values[:-1])]
126
127        adv_tensor = torch.tensor(advantages, dtype=torch.float32)
128        ret_tensor = torch.tensor(returns, dtype=torch.float32)
129
130        # Normalise advantages and returns so critic loss is
131        scale-invariant
132
133        # across scenarios (maritime rewards ~800, warehouse ~20).

```

```

128         Without this,
129         # the raw critic target causes v_loss explosion and pre-clip grad
130         norms
131         # of 400-900 on maritime, wasting nearly all of the grad budget.
132         adv_tensor = (adv_tensor - adv_tensor.mean()) / (adv_tensor.std()
133         + 1e-8)
134         ret_tensor = (ret_tensor - ret_tensor.mean()) / (ret_tensor.std()
135         + 1e-8)
136         return adv_tensor, ret_tensor
137
138     def clear(self):
139         self.__init__()
140
141     def train_rl_agent(agent: nn.Module, env, scenario: str,
142                       algo: str, seed: int,
143                       num_episodes: int = cfg.num_episodes,
144                       device: torch.device = torch.device('cpu'),
145                       checkpoint_dir: str = MODELS_DIR,
146                       log_every: int = 50,
147                       teacher_models: Optional[Dict[str, nn.Module]] = None)
148     -> Dict[str, Any]:
149     """Main RL training loop.
150
151     Args:
152         agent:          RL model (MAPPO / QMIX / MADDPG / DQN / HybridRL)
153         env:            gymnasium-compatible environment
154         scenario:       'warehouse' | 'search_and_rescue' |
155         'maritime_security'
156         algo:          algorithm name
157         seed:          random seed
158         num_episodes:  training episodes
159         device:        cuda or cpu
160         checkpoint_dir: where to save .pt checkpoints

```

```

156         log_every:      log interval
157         teacher_models: optional dict {algo_name: teacher_nn.Module} used
for
158             KL distillation during HybridRL training.  When
provided
159             for HybridRL, PCGrad-projected KL losses from all
4
160             teachers are added to the PPO update each epoch.
161
162     Returns:
163         results dict with rewards, completion rates, timing
164     """
165     # Reproducibility
166     repro = ReproducibilityManager(seed)
167
168     # Reward shaping (TA-RWARE delivery reward)
169     shaper = RewardShapingWrapper(scenario)
170
171     # Action masking (TA-RWARE)
172     masker = ActionMaskWrapper(MAX_TASKS, MAX_ROBOTS)
173
174     sc_cfg = SCENARIO_CONFIGS[scenario]
175     ep_max_steps = sc_cfg.max_steps
176
177     # Optimiser
178     optimizer = torch.optim.Adam(agent.parameters(), lr=cfg.learning_rate,
179                                   eps=1e-5)
180     scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
181         optimizer, T_max=num_episodes, eta_min=cfg.learning_rate * 0.1)
182
183     agent.to(device)
184     agent.train()
185
186     # AMP mixed precision - only on CUDA. Falls back cleanly on CPU.

```

```

187     # Opt-in via env var AQUA_AMP=1 (set by run.py for all modes).
188     _use_amp = (
189         os.environ.get('AQUA_AMP', '0') == '1'
190         and device.type == 'cuda'
191     )
192     scaler = torch.cuda.amp.GradScaler(enabled=_use_amp)
193
194     # DCMRTA improvement F: Dynamic IL decay schedule ---
episode-adaptive.
195     # Decay is scaled so the rate reaches 10% of its base by
ep=0.6*num_episodes,
196     # regardless of run length. This fixes the IL-dominance issue on
short runs
197     # (e.g. 1000 eps) where IL_DECAY_EXPONENT=-1e-5 (tuned for 70k+ ep
DCMRTA
198     # runs) decays by only 0.9% over the entire run.
199     import math as _math
200     _il_base    = sc_cfg.il_mix_rate
201     _il_warmup  = max(20, int(0.10 * num_episodes))
202     _il_decay_span = max(1, int(0.6 * num_episodes) - _il_warmup)
203     _il_decay    = _math.log(0.1) / _il_decay_span    # reach 10% of base by
60% of run
204     def _il_rate(ep: int) -> float:
205         if ep < _il_warmup:
206             return _il_base
207         return max(sc_cfg.il_min_rate, _il_base * _math.exp(_il_decay *
208             (ep - _il_warmup)))
209
210     # Training metrics
211     episode_rewards:    List[float] = []
212     completion_rates:   List[float] = []
213     episode_lengths:    List[int]   = []
214     policy_losses:      List[float] = []
215     value_losses:       List[float] = []

```

```

215     last_grad_norm: float = 0.0
216
217     os.makedirs(checkpoint_dir, exist_ok=True)
218     best_reward = -float('inf')
219
220     # Separate experience replay for DQN/MADDPG (Bellman TD
221     bootstrapping).
222     # Lives outside the episode loop so transitions accumulate across
223     episodes.
224     dqn_replay: ReplayBuffer = ReplayBuffer(capacity=50_000)
225
226     # DQN target network --- prevents non-stationary Q-targets that cause
227     # reward oscillation and collapse once IL fades. Hard-copied every
228     # _dqn_target_every episodes (~15 times over the run).
229     import copy as _copy
230     _dqn_target = _copy.deepcopy(agent).to(device).eval() if algo ==
231     'DQN' else None
232     _dqn_target_every = max(10, num_episodes // 15)
233
234     # QMIX target network: prevents the non-stationary TD bootstrap that
235     caused
236     # loss=420/grad=3630 at ep=200 and NaN cascade by ep=400 in warehouse
237     runs.
238     _qmix_target = _copy.deepcopy(agent).to(device).eval() if algo ==
239     'QMIX' else None
240     _qmix_target_every = max(10, num_episodes // 15)
241
242     logger.info(
243         f'[{algo}/{scenario}/seed={seed}] Starting {num_episodes}
244         episodes '
245         f'on {device} | ep_max_steps={ep_max_steps}'
246     )
247     logger.info(
248         f' Scenario: obs_radius={sc_cfg.observation_radius}m '
249         f'comm={sc_cfg.communication_range}m '

```



```

242         f'robots={sc_cfg.num_robots_min}-{sc_cfg.num_robots_max} '
243         f'max_tasks={sc_cfg.max_tasks}'
244     )
245     logger.info(
246         f' Hyperparams: LR={cfg.learning_rate} gamma={cfg.gamma} '
247         f'clip={cfg.clip_epsilon} grad_norm={cfg.max_grad_norm}
AMP={_use_amp}',
248     )
249     logger.info(
250         f' IL schedule: base={_il_base:.3f} warmup={_il_warmup}ep '
251         f'decay_span={_il_decay_span}ep (reaches 10% of base at '
252         f'{_il_warmup + _il_decay_span}ep)'
253     )
254     t_start = time.time()
255
256     # MADDPG Gumbel-softmax temperature annealing. Reads schedule from
257     # env_config.RLConfig.MADDPG_GUMBEL_TEMP_{START,END,DECAY}. Linear
decay
258     # from tau=1.0 down to tau=0.1 over MADDPG_GUMBEL_TEMP_DECAY episodes.
259     # Called once per episode so the actor's gumbel_sample uses the
current
260     # tau for the whole episode's rollout.
261     _gumbel_schedule = None
262     if algo == 'MADDPG' or algo == 'HybridRL':
263         from environment.env_config import rl_config as _env_rl
264         _gumbel_schedule = (
265             _env_rl.MADDPG_GUMBEL_TEMP_START,
266             _env_rl.MADDPG_GUMBEL_TEMP_END,
267             _env_rl.MADDPG_GUMBEL_TEMP_DECAY,
268         )
269
270     for ep in range(num_episodes):
271         # anneal MADDPG gumbel tau at the start of every episode
272         if _gumbel_schedule is not None:

```

```

273         s, e, d = _gumbel_schedule
274         if algo == 'MADDPG':
275             agent.step_gumbel_temp(ep, tau_start=s, tau_end=e,
decay_eps=d)
276         elif algo == 'HybridRL' and hasattr(agent, 'maddpg_actor'):
277             # HybridRL doesn't wrap a full MADDPG, just the actor. We
anneal
278             # via a module-level attribute so the actor's
gumbel_sample call
279             # (if any) uses the current temperature.
280             frac = min(1.0, max(0.0, ep / max(1, d)))
281             tau = s + frac * (e - s)
282             # Store tau as a buffer on the hybrid agent for
downstream use
283             if not hasattr(agent, 'gumbel_tau'):
284                 agent.register_buffer('gumbel_tau',
285                                     torch.tensor(tau,
device=device))
286             else:
287                 agent.gumbel_tau.fill_(tau)
288
289
290         # Variable robot count (1 max_robots)
291         # Curriculum: start with fewer robots, increase as training
progresses
292         # This trains a single policy that handles any team size 1-20.
293         if sc_cfg.train_variable_robots:
294             progress = ep / max(num_episodes - 1, 1)    # 0.0 → 1.0
295             # Curriculum: early training uses smaller fleets, scales to
full range
296             if progress < 0.3:
297                 max_n = max(sc_cfg.num_robots_min + 1,
sc_cfg.num_robots_max // 2)
298             elif progress < 0.6:

```

```

299         max_n = sc_cfg.num_robots_max
300     else:
301         max_n = MAX_ROBOTS
302     ep_num_robots = (seed + ep) % (max_n - sc_cfg.num_robots_min
+ 1)\
303         + sc_cfg.num_robots_min
304     ep_num_robots = max(sc_cfg.num_robots_min,
305         min(ep_num_robots, MAX_ROBOTS))
306 else:
307     ep_num_robots = sc_cfg.num_robots_max
308
309     obs, info = env.reset(seed=seed + ep)
310     # Sync ep_num_robots from what the env actually spawned
(PPG/curriculum
311     # sets this during reset; the pre-reset assignment was a no-op).
312     if hasattr(env, 'num_robots'):
313         ep_num_robots = env.num_robots
314     shaper.reset()
315     buffer = RolloutBuffer()
316
317     ep_reward      = 0.0
318     ep_steps       = 0
319     tasks_done_ep = 0
320
321     done = False
322     while not done and ep_steps < ep_max_steps:
323         # Build observations
324         local_obs_np = obs[:LOCAL_OBS_DIM].astype(np.float32)
325         global_obs_np = obs[:GLOBAL_OBS_DIM].astype(np.float32)\
326             if len(obs) >= GLOBAL_OBS_DIM else
obs[:LOCAL_OBS_DIM].astype(np.float32)
327         flat_obs_np = obs[:FLAT_OBS_DIM].astype(np.float32)\
328             if len(obs) >= FLAT_OBS_DIM else
obs.astype(np.float32)

```

```

329
330         local_t =
torch.from_numpy(local_obs_np).unsqueeze(0).to(device)
331         global_t =
torch.from_numpy(global_obs_np).unsqueeze(0).to(device)
332         flat_t =
torch.from_numpy(flat_obs_np).unsqueeze(0).to(device)
333
334         # Scenario-aware action masking (all 3 scenarios).
335         # the try/except fallback to the generic masker is removed.
336         # The scenario-aware mask is now the ONLY path. If
337         # get_scenario_action_mask raises, the job errors out and is
338         # isolated at the per-job boundary in rl_worker.
339         tasks      = env.tasks  if hasattr(env, 'tasks') else []
340         robots     = env.robots if hasattr(env, 'robots') else []
341         from utils.action_masking import get_scenario_action_mask
342         n_t        = len(tasks)
343         t_st       = np.array([1 if t.get('status') == 'AVAILABLE'
344                                else 0 for t in tasks]) if n_t else
np.array([])
345         r_asg      = np.array([r.get('current_task', -1) if
r.get('current_task') is not None
346                                else -1 for r in robots]) if robots else
np.array([-1])
347         r_pos      = (np.array([r.get('position', [0, 0]) for r in robots])
348                        if robots else np.zeros((ep_num_robots, 2)))
349         t_pos      = (np.array([t.get('location', [0, 0]) for t in tasks])
350                        if n_t else np.zeros((1, 2)))
351         t_pri      = (np.array([t.get('priority_val',
t.get('priority_level', 2))
352                                for t in tasks])
353                        if n_t else np.ones(1, dtype=int))
354         r_car      = (np.array([bool(r.get('current_payload', 0) > 0) for
r in robots])

```

```

355         if robots else np.zeros(ep_num_robots, dtype=bool))
356     sc_mask = get_scenario_action_mask(
357         scenario, n_t, t_st, r_asg, r_pos, t_pos, r_car, t_pri,
358         obs_radius_m=sc_cfg.observation_radius,
359         comm_range_m=sc_cfg.communication_range)
360     # Build ACTION_DIM-sized mask with MAX_TASKS stride so
indices match
361     # agent action decoding: robot_id = action // MAX_TASKS,
task_id = action % MAX_TASKS.
362     # Previously used n_t stride which misaligned every valid
action index.
363     mask_np = np.zeros(ACTION_DIM, dtype=np.float32)
364     mask_np[ACTION_DIM - 1] = 1.0    # no-op at ACTION_DIM-1 is
always valid
365     for ri in range(sc_mask.shape[0]):
366         for ti in range(n_t):
367             if sc_mask[ri, ti + 1]:
368                 flat_idx = ri * MAX_TASKS + ti
369                 if flat_idx < ACTION_DIM - 1:
370                     mask_np[flat_idx] = 1.0
371     mask = torch.from_numpy(mask_np).to(device)
372
373     # DCMRTA improvement F: IL mixing - with probability
il_rate(ep),
374     # use greedy nearest-robot-nearest-task action instead of
learned policy.
375     # Rate decays exponentially: base * exp(-1e-5 * ep)
376     _cur_il = _il_rate(ep)
377     _use_il = (_cur_il > 0 and
378               np.random.rand() < _cur_il and
379               hasattr(env, 'tasks') and hasattr(env, 'robots'))
380     if _use_il:
381         # Find the closest ROBOT-TASK pair across the full swarm.
382         # For RL-only scenarios (warehouse) prefer idle robots so

```

```

we don't
383         # overwrite an in-progress assignment and corrupt task
state.
384         _sc_il = SCENARIO_CONFIGS.get(scenario)
385         _rl_only_il = _sc_il is not None and not
_sc_il.auto_reassign_on_complete
386         _r_targets = getattr(env, 'robot_targets', {}) or {}
387         _all_robots = [r for r in (env.robots or [])
388                         if r.get('operational_status') !=
ROBOT_STATUS_FAILED
389                         and 'position' in r]
390         if _rl_only_il:
391             _idle = [r for r in _all_robots
392                     if _r_targets.get(r.get('id', -1)) is None]
393             _robots = _idle if _idle else _all_robots
394         else:
395             _robots = _all_robots
396             _tasks = [t for t in (env.tasks or [])
397                       if t.get('status') == 'AVAILABLE'
398                       and 'location' in t]
399         if _robots and _tasks:
400             # Find the (robot, task) pair with minimum distance
401             _best = None
402             _best_d = float('inf')
403             for _ri, _r in enumerate(_robots):
404                 _rpos = np.array(_r.get('position', [0., 0.]))[:2]
405                 for _ti, _t in enumerate(_tasks):
406                     _tpos = np.array(_t.get('location', [0.,
0.]))[:2]
407                     _d = float(np.linalg.norm(_rpos - _tpos))
408                     if _d < _best_d:
409                         _best_d = _d
410                         _best = (_ri, _ti, _r, _t)
411             if _best is not None:

```

```

412         _ri, _ti, _r, _t = _best
413         _task_id = _t.get('id', _ti)
414         _robot_id = _r.get('id', _ri)
415         _try_action = _robot_id * MAX_TASKS + _task_id
416         action_int = int(min(_try_action, ACTION_DIM - 2))
417     else:
418         action_int = 0
419     # Step first so we have the real reward before
storing to buffer
420     next_obs, raw_reward, terminated, truncated,
step_info = env.step(action_int)
421     done = terminated or truncated
422     shaped_reward = shaper.shape(raw_reward, step_info,
423                                   env.robots if
hasattr(env, 'robots') else [],
424                                   env.tasks if
hasattr(env, 'tasks') else [])
425     buffer.add(local_obs_np, global_obs_np, flat_obs_np,
426               action_int, 0.0, shaped_reward, 0.0,
float(done),
427               mask.cpu().numpy())
428     if algo in ('DQN', 'MADDPG', 'QMIX'):
429         next_flat =
(next_obs[:FLAT_OBS_DIM].astype(np.float32)
430         if len(next_obs) >= FLAT_OBS_DIM
431         else next_obs.astype(np.float32))
432     dqn_replay.add(flat_obs_np, action_int,
shaped_reward,
433                   next_flat, float(done))
434     ep_reward += shaped_reward
435     ep_steps += 1
436     obs = next_obs
437     continue
438

```

```

439         # Get action from agent
440         # Epsilon-greedy exploration for value-based algos
441         # Decays from 0.9 0.05 over the first 30% of episodes. Without
442         # this, DQN/QMIX/MADDPG converge to "always pick highest-Q
valid
443         # action" from episode 0 - which is usually the same action
every
444         # step, preventing discovery of tasks further away from the
swarm
445         # centre. Samples uniformly over VALID actions (per mask) so
446         # exploration isn't wasted on env-rejected actions.
447         _eps_pct = min(1.0, ep / max(1, int(0.3 * num_episodes)))
448         _eps_val = 0.9 * (1.0 - _eps_pct) + 0.05 * _eps_pct
449         _do_eps = (algo in ('DQN', 'MADDPG', 'QMIX')
450                    and np.random.rand() < _eps_val)
451
452         with torch.no_grad():
453             if _do_eps:
454                 # Random VALID action (respects mask)
455                 _valid = mask.cpu().numpy() > 0
456                 _vidx = np.where(_valid)[0]
457                 if len(_vidx) > 0:
458                     action =
torch.tensor(int(np.random.choice(_vidx)), device=device)
459                 else:
460                     # Fallback: use agent (shouldn't happen if masker
works)
461                     action = agent.get_action(flat_t, mask=mask)\
462                             if algo in ('MADDPG', 'DQN')\
463                             else torch.tensor(0, device=device)
464                     log_prob = torch.tensor(0.0); value =
torch.tensor(0.0)
465                 elif algo == 'MAPPO':
466                     action, log_prob, value = agent.get_action(

```



```

467         local_t, global_t, mask=mask, deterministic=False)
468     elif algo == 'QMIX':
469         q = agent.get_agent_q(local_t, 0, mask=mask)
470         probs = torch.softmax(q, dim=-1)
471         if torch.isnan(probs).any():
472             valid = (mask > 0.5).unsqueeze(0).float()
473             probs = valid / valid.sum(dim=-1,
keepdim=True).clamp(min=1.0)
474         dist = torch.distributions.Categorical(probs)
475         action = dist.sample()
476         log_prob = dist.log_prob(action)
477         value = torch.tensor(0.0)
478     elif algo in ('MADDPG', 'DQN'):
479         action = agent.get_action(flat_t, mask=mask)
480         log_prob = torch.tensor(0.0)
481         value = torch.tensor(0.0)
482     elif algo == 'HybridRL':
483         action, log_prob, value, _ = agent.get_action(
484             local_t, flat_t, mask=mask, step=ep *
cfg.max_steps + ep_steps)
485     else:
486         action = torch.tensor(0)
487         log_prob = torch.tensor(0.0)
488         value = torch.tensor(0.0)
489
490     action_int = int(action.item()) if torch.is_tensor(action)
else int(action)
491
492     # Environment step
493     next_obs, raw_reward, terminated, truncated, step_info =
env.step(action_int)
494     done = terminated or truncated
495
496     # Shaped reward (TA-RWARE delivery reward)

```

```

497         robots = env.robots if hasattr(env, 'robots') else []
498         tasks = env.tasks if hasattr(env, 'tasks') else []
499         shaped_reward = shaper.shape(raw_reward, step_info, robots,
tasks)

500
501         if logger.isEnabledFor(logging.DEBUG):
502             tasks_just_done = (step_info or
503         {}).get('tasks_completed', 0)
504             if tasks_just_done:
505                 logger.debug(
506                     f'[{algo}/{scenario}/s{seed}] ep={ep+1}
507         step={ep_steps} '
508                     f'completed {tasks_just_done} task(s) '
509                     f'raw={raw_reward:.4f}
510         shaped={shaped_reward:.4f}'
511                 )
512
513         # Store in buffer
514         buffer.add(local_obs_np, global_obs_np, flat_obs_np,
515         action_int,
516         float(log_prob.item()) if
517         torch.is_tensor(log_prob) else 0.0,
518         shaped_reward,
519         float(value.item()) if torch.is_tensor(value) else
520         0.0,
521         float(done), mask.cpu().numpy())
522
523         # Also push to replay buffer for DQN/MADDPG/QMIX TD
524         bootstrapping
525         if algo in ('DQN', 'MADDPG', 'QMIX'):
526             next_flat = (next_obs[:FLAT_OBS_DIM].astype(np.float32)
527                 if len(next_obs) >= FLAT_OBS_DIM else
528             next_obs.astype(np.float32))
529             dq_n_replay.add(flat_obs_np, action_int, shaped_reward,

```

```

next_flat, float(done))

523
524         ep_reward += shaped_reward
525         ep_steps  += 1
526         obs = next_obs
527
528         # PPO / policy gradient update
529         p_loss, v_loss = 0.0, 0.0
530
531         if len(buffer.rewards) >= 2 and algo in ('MAPPO', 'HybridRL'):
532             adv, ret = buffer.compute_returns()
533             adv = adv.to(device)
534             ret = ret.to(device)
535
536             local_t =
torch.from_numpy(np.array(buffer.local_obs)).to(device)
537             global_t =
torch.from_numpy(np.array(buffer.global_obs)).to(device)
538             flat_t =
torch.from_numpy(np.array(buffer.flat_obs)).to(device)
539             act_t = torch.tensor(buffer.actions,
dtype=torch.long).to(device)
540             old_lp = torch.tensor(buffer.log_probs,
dtype=torch.float32).to(device)
541             _raw_masks = [m if m is not None else np.ones(ACTION_DIM,
dtype=np.float32)
542                         for m in buffer.masks]
543             masks_t = torch.from_numpy(np.array(_raw_masks,
dtype=np.float32)).to(device)
544
545             for _ in range(4): # PPO epochs
546                 with torch.cuda.amp.autocast(enabled=_use_amp):
547                     log_prob, entropy, value = agent.evaluate(
548                         local_t, global_t, act_t, mask=masks_t)

```

```

549
550         ratio = torch.exp(log_prob - old_lp)
551         surr1 = ratio * adv
552         surr2 = torch.clamp(ratio, 1 - cfg.clip_epsilon,
553                             1 + cfg.clip_epsilon) *
adv
554         actor_loss = -torch.min(surr1, surr2).mean()
555         critic_loss = cfg.value_loss_coef * F.mse_loss(value,
ret)
556         ent_loss = -cfg.entropy_coef * entropy.mean()
557         loss = actor_loss + critic_loss + ent_loss
558
559         # KL distillation for HybridRL: mean KL(student //
teacher) added to
560         # the PPO loss so both objectives are optimised in one
backward pass.
561         if algo == 'HybridRL' and teacher_models:
562             kl_terms = []
563             for t_name, t_model in teacher_models.items():
564                 t_model.eval()
565                 try:
566                     with torch.no_grad():
567                         t_logits = t_model(local_t, flat_t) \
568                             if hasattr(t_model, 'forward') else
None
569                         if t_logits is not None and
isinstance(t_logits, torch.Tensor):
570                             t_probs = torch.softmax(t_logits.float(),
dim=-1).clamp(1e-8, 1.0)
571                             s_logits = agent.mappo_actor(local_t,
None)
572                             s_log_probs =
torch.log_softmax(s_logits.float(), dim=-1)
573                             kl = (t_probs * (t_probs.log() -

```

```

s_log_probs)).sum(dim=-1).mean()
574         kl_terms.append(kl)
575     except Exception:
576         pass
577     if kl_terms:
578         loss = loss + sum(kl_terms) / len(kl_terms)
579
580     if torch.isnan(loss) or torch.isinf(loss):
581         logger.warning(
582             f'[{algo}/{scenario}/s{seed}] PPO loss NaN/Inf ',
583             f'(actor={actor_loss.item():.4f} ',
584             f'critic={critic_loss.item():.4f}) ',
585             f'at ep={ep+1}, skipping epoch')
586         break
587     optimizer.zero_grad()
588     # OOM-retry-halving try/except removed. Any RuntimeError
589     # (including OOM) now propagates to rl_worker's per-job
590     # try/except and marks just this one job errored. The old
591     # silent batch-halving was also buggy: the inner
try/except
592     # swallowed shrink_config_batches_on_oom errors too.
593     scaler.scale(loss).backward()
594     scaler.unscale_(optimizer)
595     _gn = nn.utils.clip_grad_norm_(agent.parameters(),
cfg.max_grad_norm)
596     last_grad_norm = float(_gn.item()) if
torch.is_tensor(_gn) else float(_gn)
597     scaler.step(optimizer)
598     scaler.update()
599
600     p_loss += actor_loss.item()
601     v_loss += critic_loss.item()
602
603     elif algo == 'QMIX' and len(dqn_replay) >= cfg.batch_size:

```

```

604         # QMIX TD loss: individual Q routed through monotonic mixing
        network
605         obs_b, act_b, rew_b, nobs_b, done_b =
dqn_replay.sample(cfg.batch_size)
606         local_t = torch.from_numpy(obs_b[:,
:LOCAL_OBS_DIM]).to(device)
607         nlocal_t = torch.from_numpy(nobs_b[:,
:LOCAL_OBS_DIM]).to(device)
608         state_t = torch.from_numpy(obs_b[:,
:GLOBAL_OBS_DIM]).to(device)
609         nstate_t = torch.from_numpy(nobs_b[:,
:GLOBAL_OBS_DIM]).to(device)
610         act_t = torch.tensor(act_b, dtype=torch.long).to(device)
611         rews_t = torch.tensor(rew_b,
dtype=torch.float32).to(device)
612         dones_t = torch.tensor(done_b,
dtype=torch.float32).to(device)
613
614         with torch.cuda.amp.autocast(enabled=_use_amp):
615             q_i = agent.get_agent_q(local_t, 0)
616             q_chosen = q_i[torch.arange(len(act_t)),
act_t].unsqueeze(1) # (B, 1)
617             q_total = agent.mixer(q_chosen, state_t).squeeze(-1)
        # (B,)
618
619         with torch.no_grad():
620             _tgt = _qmix_target if _qmix_target is not None else
agent
621             q_next_i = _tgt.get_agent_q(nlocal_t, 0)
622             q_next_max = q_next_i.max(dim=-1,
keepdim=True).values # (B, 1)
623             q_total_next = _tgt.mixer(q_next_max,
nstate_t).squeeze(-1)
624

```

```

625         td_target = rews_t + cfg.gamma * q_total_next * (1.0 -
dones_t)
626         qmix_loss = F.smooth_l1_loss(q_total, td_target.detach())
627
628         if torch.isnan(qmix_loss) or torch.isinf(qmix_loss):
629             logger.warning(
630                 f'[{algo}/{scenario}/s{seed}] QMIX loss NaN/Inf '
631                 f'at ep={ep+1}, skipping update')
632         else:
633             optimizer.zero_grad()
634             scaler.scale(qmix_loss).backward()
635             scaler.unscale_(optimizer)
636             _gn = nn.utils.clip_grad_norm_(agent.parameters(),
cfg.max_grad_norm)
637             last_grad_norm = float(_gn.item()) if
torch.is_tensor(_gn) else float(_gn)
638             scaler.step(optimizer)
639             scaler.update()
640             p_loss = qmix_loss.item()
641
642         elif algo in ('MADDPG', 'DQN') and len(dqn_replay) >=
cfg.batch_size:
643             # Q-learning update with Bellman TD bootstrapping from replay
buffer
644             obs_b, act_b, rew_b, nobs_b, done_b =
dqn_replay.sample(cfg.batch_size)
645             flat_t = torch.from_numpy(obs_b).to(device)
646             nflat_t = torch.from_numpy(nobs_b).to(device)
647             act_t = torch.tensor(act_b, dtype=torch.long).to(device)
648             rews_t = torch.tensor(rew_b, dtype=torch.float32).to(device)
649             dones_t = torch.tensor(done_b,
dtype=torch.float32).to(device)
650
651             if algo == 'DQN':

```

```

652         with torch.cuda.amp.autocast(enabled=_use_amp):
653             q_vals =
agent.q_values(flat_t)[torch.arange(len(act_t)), act_t]
654             with torch.no_grad():
655                 _tgt_net = _dqn_target if _dqn_target is not None
else agent
656                 q_next =
_tgt_net.q_values(nflat_t).max(dim=-1).values
657                 q_tgt = rews_t + cfg.gamma * q_next * (1.0 - dones_t)
658                 td_loss = F.smooth_l1_loss(q_vals, q_tgt.detach())
659                 optimizer.zero_grad()
660                 scaler.scale(td_loss).backward()
661                 scaler.unscale_(optimizer)
662                 _gn = nn.utils.clip_grad_norm_(agent.parameters(),
cfg.max_grad_norm)
663                 last_grad_norm = float(_gn.item()) if
torch.is_tensor(_gn) else float(_gn)
664                 scaler.step(optimizer)
665                 scaler.update()
666                 p_loss = td_loss.item()
667
668             elif algo == 'MADDPG':
669                 # TD update on actor logits (discrete Gumbel-softmax
MADDPG).
670                 # Treats tanh-clipped logits as Q-values; bootstraps from
next-state
671                 # greedy logit without a separate target network.
672                 with torch.cuda.amp.autocast(enabled=_use_amp):
673                     logits = agent.actor.logits(flat_t)
674                     q_chosen = logits[torch.arange(len(act_t)), act_t]
675                     with torch.no_grad():
676                         q_next =
agent.actor.logits(nflat_t).max(dim=-1).values
677                     td_target = rews_t + cfg.gamma * q_next * (1.0 -

```



```

dones_t)
678         td_loss    = F.smooth_l1_loss(q_chosen,
td_target.detach())
679         optimizer.zero_grad()
680         scaler.scale(td_loss).backward()
681         scaler.unscale_(optimizer)
682         _gn = nn.utils.clip_grad_norm_(agent.parameters(),
cfg.max_grad_norm)
683         last_grad_norm = float(_gn.item()) if
torch.is_tensor(_gn) else float(_gn)
684         scaler.step(optimizer)
685         scaler.update()
686         p_loss = td_loss.item()
687
688         # Hard-copy online weights to target networks every N episodes
689         if _dqn_target is not None and (ep + 1) % _dqn_target_every == 0:
690             _dqn_target.load_state_dict(agent.state_dict())
691         if _qmix_target is not None and (ep + 1) % _qmix_target_every ==
0:
692             _qmix_target.load_state_dict(agent.state_dict())
693
694         scheduler.step()
695
696         # Metrics --- count completed tasks from env state after episode
ends
697         if hasattr(env, 'tasks'):
698             tasks_done_ep = sum(1 for t in env.tasks
699                               if t.get('status') ==
TASK_STATUS_COMPLETE)
700             n_tasks = len(env.tasks)
701         else:
702             tasks_done_ep = 0
703             n_tasks = 1
704         comp_rate = tasks_done_ep / max(n_tasks, 1)

```

```

705         if hasattr(env, 'record_episode_outcome'):
706             env.record_episode_outcome(comp_rate)
707         episode_rewards.append(ep_reward)
708         completion_rates.append(comp_rate)
709         episode_lengths.append(ep_steps)
710         if p_loss: policy_losses.append(p_loss)
711         if v_loss: value_losses.append(v_loss)
712
713         # Logging
714         if (ep + 1) % log_every == 0:
715             window = min(log_every, len(episode_rewards))
716             mean_r = float(np.mean(episode_rewards[-window:]))
717             mean_c = float(np.mean(completion_rates[-window:])) * 100
718             mean_len = float(np.mean(episode_lengths[-window:]))
719             mean_pl = float(np.mean(policy_losses[-window:])) if
policy_losses else 0.0
720             mean_vl = float(np.mean(value_losses[-window:])) if
value_losses else 0.0
721             _cur_eps_frac = min(1.0, ep / max(1, int(0.3 * num_episodes)))
722             _cur_eps_val = 0.9 * (1.0 - _cur_eps_frac) + 0.05 *
_cur_eps_frac
723             _cur_il_log = _il_rate(ep)
724             elapsed = (time.time() - t_start) / 60
725             logger.info(
726                 f'[{algo}/{scenario}/s{seed}] ep={ep+1:5d}/{num_episodes}
,
727                 f'robots={ep_num_robots:2d}
steps={mean_len:.0f}/{ep_max_steps} ',
728                 f'reward={mean_r:9.3f} compl={mean_c:5.1f}% ',
729                 f'p_loss={mean_pl:.5f} v_loss={mean_vl:.5f} ',
730                 f'grad={last_grad_norm:.3f} eps={_cur_eps_val:.3f} ',
731                 f'IL={_cur_il_log:.3f} best={best_reward:.3f} ',
732                 f'elapsed={elapsed:.1f}min'
733             )

```

```

734         # Warn if reward shows no improvement over recent 3 windows
and
735         # best_reward hasn't grown since the prior interval
(suppresses
736         # false positives on agents that are still finding new best
episodes).
737         if len(episode_rewards) >= log_every * 3:
738             recent = float(np.mean(episode_rewards[-log_every:]))
739             older = float(np.mean(episode_rewards[-log_every *
3:-log_every * 2]))
740             best_at_prev =
float(np.max(episode_rewards[:-log_every])) \
741                 if len(episode_rewards) > log_every else
-float('inf')
742             if recent <= older and abs(older) > 0.1 and best_reward
<= best_at_prev:
743                 logger.warning(
744                     f'[{algo}/{scenario}/s{seed}] STUCK? reward not
improving: '
745                     f'older_window={older:.3f}
recent_window={recent:.3f} '
746                     f'delta={recent - older:.3f} --- check mask,
reward, env reset'
747                 )
748
749         # Checkpoint best model
750         if ep_reward > best_reward:
751             best_reward = ep_reward
752             ckpt_path = os.path.join(checkpoint_dir,
753
f'{algo}_{scenario}_seed{seed}_best.pt')
754             torch.save({
755                 'episode': ep,
756                 'model_state': agent.state_dict(),

```

```

757         'optimizer_state': optimizer.state_dict(),
758         'reward': best_reward,
759     }, ckpt_path)
760     logger.debug(
761         f'[{algo}/{scenario}/s{seed}] ep={ep+1} new best '
762         f'reward={best_reward:.3f} ckpt saved -> {ckpt_path}'
763     )
764
765     # CUDA cleanup
766     if (ep + 1) % 100 == 0 and torch.cuda.is_available():
767         torch.cuda.empty_cache()
768
769     # Save final model
770     final_path = os.path.join(checkpoint_dir,
771         f'{algo}_{scenario}_seed{seed}_final.pt')
772     torch.save(agent.state_dict(), final_path)
773
774     total_time = (time.time() - t_start) / 60
775     final_comp = float(np.mean(completion_rates[-100:])) * 100
776     mean_rew_final = float(np.mean(episode_rewards[-100:])) if
777     episode_rewards else 0.0
778     logger.info(
779         f'[{algo}/{scenario}/seed={seed}] Training complete: '
780         f'{total_time:.1f}min best_reward={best_reward:.3f} '
781         f'final_completion={final_comp:.1f}% '
782         f'mean_reward_last100={mean_rew_final:.3f}'
783     )
784     if final_comp < 5.0 and num_episodes >= 100:
785         logger.warning(
786             f'[{algo}/{scenario}/seed={seed}] Low final completion
787             ({final_comp:.1f}%) '
788             f'after {num_episodes} episodes --- check reward shaping,
789             action masking, '
790             f'or curriculum difficulty'

```

```

787         )
788
789     return {
790         'algo':            algo,
791         'scenario':        scenario,
792         'seed':            seed,
793         'rewards':         episode_rewards,
794         'completion_rates': completion_rates,
795         'episode_lengths': episode_lengths,
796         'policy_losses':   policy_losses,
797         'value_losses':    value_losses,
798         'best_reward':     best_reward,
799         'final_completion': float(np.mean(completion_rates[-100:])),
800         'training_time_min': total_time,
801         'model_path':      final_path,
802     }
803
804
805     # Avoid NameError for F
806     import torch.nn.functional as F
807
808
809     # the train_rl_agent_with_real_env wrapper has been deleted. It was a
810     # 25-line stub that existed only to forward to train_rl_agent after a
811     # historical detour through environment.training_loop.fallback_train_rl
812     # (which was also deleted ). run.py's rl_worker now calls
813     # train_rl_agent directly.

```

Listing B.17: RL trainer with PPO clipped surrogate, GAE advantage estimation, mixed-precision casting and cosine learning rate schedule.

```

1  """
2  training/diffusion_trainer.py
3  =====

```

```

4 | Diffusion model training loop for all five diffusion architectures.
5 |
6 | Trajectory representation: 8 B-spline control points instead of dense
7 | waypoints --- this reduces output dimensionality and guarantees smoothness
8 | without an explicit smoothness loss term (MPD-splines).
9 |
10 | Training data comes exclusively from RL rollouts that exceed a quality
11 | threshold (min_traj_quality), preventing the denoiser from learning
12 | sub-optimal paths. Flip and rotation augmentation is applied to each
13 | trajectory to improve data efficiency in the limited-rollout regime.
14 | """
15 | import os
16 | import time
17 | import torch
18 | import torch.nn as nn
19 | import numpy as np
20 | from pathlib import Path
21 | from typing import Dict, Any, List, Tuple, Optional
22 | from torch.utils.data import Dataset, DataLoader
23 |
24 | from config.base_config import (
25 |     DiffusionConfig, SCENARIO_CONFIGS, MAX_ROBOTS, MAX_TASKS,
26 |     GLOBAL_OBS_DIM, MODELS_DIR, RESULTS_DIR,
27 |     TASK_STATUS_COMPLETE,
28 | )
29 | from utils.reproducibility import ReproducibilityManager
30 | from utils.bspline import smoothness_loss
31 |
32 | cfg = DiffusionConfig()
33 |
34 |
35 | class LimitsNormalizer:
36 |     """
37 |     Maps [x_min, x_max] [-1, 1] for each dimension.

```

```

38 (MMD/MPD-splines normalization.py)
39 Used to normalise B-spline control points before diffusion training.
40 """
41     def __init__(self, X: 'torch.Tensor', eps: float = 1e-6):
42         import torch
43         self.mins = X.flatten(0, -2).min(dim=0).values
44         self.maxs = X.flatten(0, -2).max(dim=0).values
45         self.eps = eps
46
47     def normalize(self, x):
48         import torch
49         rng = (self.maxs - self.mins).clamp(min=self.eps)
50         return 2.0 * (x - self.mins) / rng - 1.0
51
52     def unnormailze(self, x):
53         import torch
54         rng = (self.maxs - self.mins).clamp(min=self.eps)
55         return (x + 1.0) / 2.0 * rng + self.mins
56
57
58 class SafeLimitsNormalizer(LimitsNormalizer):
59     """
60     Like LimitsNormalizer but handles constant dimensions (min==max)
61     by adding eps - prevents NaN when a trajectory dim doesn't vary.
62     (MMD normalization.py - SafeLimitsNormalizer)
63     """
64     def __init__(self, X: 'torch.Tensor', eps: float = 1.0):
65         super().__init__(X, eps=1e-6)
66         import torch
67         constant = (self.maxs - self.mins) < 1e-6
68         self.mins[constant] -= eps
69         self.maxs[constant] += eps
70
71

```

```

72
73 class TrajectoryDataset(Dataset):
74     """
75     Dataset of trajectories collected from trained RL agent.
76     Each sample: (obs, start_pos, goal_pos, waypoints)
77     """
78
79     def __init__(self, trajectories: List[Dict]):
80         self.data = trajectories
81
82     def __len__(self):
83         return len(self.data)
84
85     def __getitem__(self, idx):
86         item = self.data[idx]
87         return (torch.tensor(item['obs'], dtype=torch.float32),
88                 torch.tensor(item['start_pos'], dtype=torch.float32),
89                 torch.tensor(item['goal_pos'], dtype=torch.float32),
90                 torch.tensor(item['waypoints'], dtype=torch.float32))
91
92     @classmethod
93     def augment(cls, traj_list: List[Dict]) -> List[Dict]:
94         """
95         Trajectory augmentation from MPD:
96         - Mirror (flip x axis)
97         - Rotate by 90/180/270 degrees
98
99         4× data multiplication from geometric augmentations.
100        """
101        augmented = []
102        for item in traj_list:
103            augmented.append(item)
104            wp = np.array(item['waypoints'])
105

```



```

106         # Flip x
107         wp_fx = wp.copy(); wp_fx[:, 0] = -wp_fx[:, 0]
108         augmented.append(**item, 'waypoints': wp_fx.tolist(),
109                             'start_pos': [-item['start_pos'][0],
item['start_pos'][1]],
110                             'goal_pos': [-item['goal_pos'][0],
item['goal_pos'][1]]})
111
112         # Flip y
113         wp_fy = wp.copy(); wp_fy[:, 1] = -wp_fy[:, 1]
114         augmented.append(**item, 'waypoints': wp_fy.tolist(),
115                             'start_pos': [item['start_pos'][0],
- item['start_pos'][1]],
116                             'goal_pos': [item['goal_pos'][0],
- item['goal_pos'][1]]})
117
118         # Rotate 90° CCW
119         wp_r90 = np.stack([-wp[:, 1], wp[:, 0]], axis=-1)
120         augmented.append(**item, 'waypoints': wp_r90.tolist(),
121                             'start_pos': [-item['start_pos'][1],
item['start_pos'][0]],
122                             'goal_pos': [-item['goal_pos'][1],
item['goal_pos'][0]]})
123
124         # Rotate 180°
125         wp_rot = -wp
126         augmented.append(**item, 'waypoints': wp_rot.tolist(),
127                             'start_pos': [-item['start_pos'][0],
- item['start_pos'][1]],
128                             'goal_pos': [-item['goal_pos'][0],
- item['goal_pos'][1]]})
129
130         return augmented
131

```

```

132
133 def collect_trajectories(rl_agent: nn.Module, env, scenario: str,
134                         n_episodes: int = 200,
135                         device: torch.device = torch.device('cpu'),
136                         seed: int = 42) -> List[Dict]:
137     """
138     Run the trained RL agent and collect (obs, start, goal, waypoints)
139     tuples.
140
141     Only high-quality trajectories (task completion > 0) are kept.
142
143     NOTE: n_episodes is the default - run.py overrides via
144           DiffusionConfig.traj_collect_episodes
145           so modes can shrink it cheaply.
146     """
147     # Allow mode override via class attr set by run.py
148     n_episodes = int(getattr(cfg, 'traj_collect_episodes', n_episodes))
149     from config.base_config import LOCAL_OBS_DIM, FLAT_OBS_DIM
150     from utils.action_masking import ActionMaskWrapper
151
152     sc_cfg = SCENARIO_CONFIGS[scenario]
153     masker = ActionMaskWrapper(MAX_TASKS, MAX_ROBOTS)
154     trajectories = []
155
156     rl_agent.eval()
157     rl_agent.to(device)
158
159     # Identify algo once per call - cheaper than probing every step
160     _algo = getattr(rl_agent, 'name', rl_agent.__class__.__name__)
161
162     with torch.no_grad():
163         for ep in range(n_episodes):
164             obs, _ = env.reset(seed=seed + ep)
165             robot_trajs = {i: {'positions': [], 'obs_list': [], 'tasks':

```

```

163         for i in range(MAX_ROBOTS)}
164     done = False
165     steps = 0
166
167     while not done and steps < cfg.T * 2:
168         local_obs = obs[:LOCAL_OBS_DIM].astype(np.float32)
169         global_obs = obs[:GLOBAL_OBS_DIM].astype(np.float32)\
170             if len(obs) >= GLOBAL_OBS_DIM else
171         local_obs.copy()
172         flat_obs = obs[:FLAT_OBS_DIM].astype(np.float32)\
173             if len(obs) >= FLAT_OBS_DIM else
174         obs.astype(np.float32)
175
176         lt = torch.from_numpy(local_obs ).unsqueeze(0).to(device)
177         gt = torch.from_numpy(global_obs).unsqueeze(0).to(device)
178         ft = torch.from_numpy(flat_obs ).unsqueeze(0).to(device)
179         mask = masker.get_mask(
180             getattr(env, 'tasks', []),
181             getattr(env, 'num_robots', MAX_ROBOTS),
182             flat_obs).to(device)
183
184         # Per-agent dispatch - each agent has a different
185         get_action
186         # signature. removed the outer try/except that fell back
187         to
188         # a random valid action. If dispatch fails, the diffusion
189         job
190         # errors and is isolated at the per-job boundary in
191         diff_worker.
192         # An unknown _algo now raises instead of silently using
193         the
194         # TorchScript forward path (which only returned (robot_id,
195         # task_id), not a raw action index and was masking bugs).
196         if _algo == 'MAPPO':

```

```

190         action_t, _, _ = rl_agent.get_action(
191             lt, gt, mask=mask, deterministic=True)
192     elif _algo == 'QMIX':
193         q = rl_agent.get_agent_q(lt, 0, mask=mask)
194         action_t = torch.argmax(q, dim=-1)
195     elif _algo == 'MADDPG':
196         action_t = rl_agent.get_action(
197             ft, mask=mask, deterministic=True)
198     elif _algo == 'DQN':
199         # DQN signature is (flat_obs, eps=0.0, mask=None) -
200         NO 'deterministic'
201         action_t = rl_agent.get_action(ft, eps=0.0, mask=mask)
202     elif _algo == 'HybridRL':
203         action_t, _, _, _ = rl_agent.get_action(
204             lt, ft, mask=mask, deterministic=True, step=1)
205     else:
206         raise ValueError(
207             f'Unknown RL algo {_algo!r} in diffusion
208 trajectory '
209             f'collection. Valid: MAPPO / QMIX / MADDPG / DQN
210 / HybridRL.')
211     action_int = (int(action_t.item()) if
212 torch.is_tensor(action_t)
213                 else int(action_t))
214
215     # Record robot positions
216     robots = getattr(env, 'robots', [])
217     for i, robot in enumerate(robots[:MAX_ROBOTS]):
218         pos = np.asarray(robot.get('position', [0, 0]))
219         robot_trajs[i]['positions'].append(pos.tolist())
220         robot_trajs[i]['obs_list'].append(
221             obs[:GLOBAL_OBS_DIM].astype(np.float32).tolist())
222
223     obs, _, terminated, truncated, _ = env.step(action_int)

```

```

220         done = terminated or truncated
221         steps += 1
222
223         # Extract trajectory samples from completed episodes
224         tasks = getattr(env, 'tasks', [])
225         completed = [t for t in tasks if t.get('status') ==
TASK_STATUS_COMPLETE]
226         # Accept episodes with completions OR sufficient traversal
steps so
227         # that early-stage agents (0 % task completion) still
contribute
228         # trajectory data for diffusion training.
229         if not completed and steps < 8:
230             continue # skip only truly degenerate (near-zero-step)
episodes
231
232         grid_half = sc_cfg.grid_half
233         robots = getattr(env, 'robots', [])
234
235         for i, robot in enumerate(robots[:MAX_ROBOTS]):
236             positions = robot_trajs[i]['positions']
237             obs_list = robot_trajs[i]['obs_list']
238             if len(positions) < 4:
239                 continue
240
241             wp = np.array(positions) / grid_half # normalise
242             # Smooth and subsample to cfg.bspline_n_eval points
243             if len(wp) > cfg.bspline_n_eval:
244                 idx = np.linspace(0, len(wp) - 1,
cfg.bspline_n_eval).astype(int)
245                 wp = wp[idx]
246             elif len(wp) < cfg.bspline_n_eval:
247                 # Interpolate
248                 from scipy.interpolate import interp1d

```

```

249         t_orig = np.linspace(0, 1, len(wp))
250         t_new = np.linspace(0, 1, cfg.bspline_n_eval)
251         fx = interp1d(t_orig, wp[:, 0]); fy =
interp1d(t_orig, wp[:, 1])
252         wp = np.stack([fx(t_new), fy(t_new)], axis=-1)
253
254         # Use middle obs as conditioning
255         mid = len(obs_list) // 2
256         obs_sample = obs_list[mid]
257
258         trajectories.append({
259             'obs':      obs_sample,
260             'start_pos': wp[0].tolist(),
261             'goal_pos':  wp[-1].tolist(),
262             'waypoints': wp.tolist(),
263         })
264
265     print(f' Collected {len(trajectories)} trajectories from
{n_episodes} episodes')
266     return trajectories
267
268
269 def train_diffusion_model(diff_model: nn.Module,
270                           rl_agent: nn.Module,
271                           env,
272                           scenario: str,
273                           diff_algo: str,
274                           rl_algo: str,
275                           seed: int,
276                           num_epochs: int = cfg.num_epochs,
277                           device: torch.device = torch.device('cpu'),
278                           checkpoint_dir: str = MODELS_DIR) -> Dict[str,
Any]:
279     """

```

```

280 Train a diffusion model on trajectories from the RL agent.
281 """
282     repro = ReproducibilityManager(seed)
283
284     print(f'\n[{diff_algo}]/[{rl_algo}]/[{scenario}]/seed={seed}] '
285           f'Collecting trajectories...')
286
287     # Collect trajectories from RL agent (same env, same seed)
288     trajs = collect_trajectories(rl_agent, env, scenario,
289                                 n_episodes=200, device=device,
290                                 seed=seed)
291
292     if len(trajs) < cfg.batch_size:
293         # Augment if too few
294         trajs = TrajectoryDataset.augment(trajs)
295         print(f' Augmented to {len(trajs)} trajectories')
296
297     if len(trajs) < 4:
298         print(f' WARNING: only {len(trajs)} trajectories --- diffusion
299               training skipped')
300         return {
301             'diff_algo': diff_algo, 'rl_algo': rl_algo,
302             'scenario': scenario, 'seed': seed,
303             'status': 'skipped',
304             'reason': f'insufficient trajectories ({len(trajs)} < 4)',
305             'diff_loss': [], 'best_loss': float('inf'),
306             'n_trajectories': len(trajs),
307             'model_path': None,
308         }
309
310     dataset = TrajectoryDataset(trajs)
311     loader = DataLoader(dataset, batch_size=cfg.batch_size,
312                         shuffle=True, drop_last=True, num_workers=0)

```

```

312     optimizer = torch.optim.Adam(diff_model.parameters(),
313                                   lr=cfg.learning_rate)
314     scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
315         optimizer, T_max=num_epochs, eta_min=cfg.learning_rate * 0.1)
316
317     diff_model.to(device)
318     diff_model.train()
319
320     # AMP mixed precision (opt-in via AQUA_AMP=1, set by run.py).
321     # Diffusion training is especially noise-tolerant - AMP gives 1.5-2x
322     on A6000.
323     _use_amp = (
324         os.environ.get('AQUA_AMP', '0') == '1'
325         and device.type == 'cuda'
326     )
327     scaler = torch.cuda.amp.GradScaler(enabled=_use_amp)
328
329     # Min-SNR loss weighting (Hang et al., "Efficient Diffusion Training
330     via Min-SNR Weighting").
331     # Enabled via AQUA_MIN_SNR=1 env var. Approximate: we don't have
332     per-sample
333     # timesteps in the wrapped training_loss, so we apply a constant
334     weight < 1
335     # to early epochs (high-noise) where the loss is hardest. This is a
336     mild
337     # version of the full per-t weighting but preserves the convergence
338     speedup.
339     _use_min_snr = os.environ.get('AQUA_MIN_SNR', '0') == '1'
340
341     epoch_losses: List[float] = []
342     best_loss = float('inf')
343     os.makedirs(checkpoint_dir, exist_ok=True)
344
345     t_start = time.time()

```



```

339     print(f' Training {diff_algo} on {len(trajs)} trajectories for
        {num_epochs} epochs')
340
341     for epoch in range(num_epochs):
342         batch_losses = []
343         # Min-SNR: down-weight the first ~25% of epochs (high noise
        regime).
344         # This mirrors the effect of per-timestep 1/SNR clamping without
        needing
345         # the model to expose its sampled t's.
346         if _use_min_snr and epoch < max(1, num_epochs // 4):
347             _loss_weight = 0.5
348         else:
349             _loss_weight = 1.0
350
351         for obs_b, start_b, goal_b, wp_b in loader:
352             obs_b = obs_b.to(device)
353             start_b = start_b.to(device)
354             goal_b = goal_b.to(device)
355             wp_b = wp_b.to(device)    # (batch, n_eval, 2)
356
357             optimizer.zero_grad()
358
359             with torch.cuda.amp.autocast(enabled=_use_amp):
360                 # Training loss (MSE noise + smoothness)
361                 if hasattr(diff_model, 'training_loss'):
362                     loss = diff_model.training_loss(wp_b, obs_b, start_b,
        goal_b)
363                 elif hasattr(diff_model, 'tdm'):
364                     # HybridDiffusion
365                     loss = diff_model.training_loss(wp_b, obs_b, start_b,
        goal_b)
366             else:
367                 continue

```

```

368
369         if torch.isnan(loss) or torch.isinf(loss):
370             continue
371
372         loss = loss * _loss_weight
373
374         scaler.scale(loss).backward()
375         scaler.unscale_(optimizer)
376         nn.utils.clip_grad_norm_(diff_model.parameters(), 5.0)
377
378         # OOM-retry-halving try/except removed (per no-fallbacks
policy).
379         # If scaler.step OOMs, it propagates to diff_worker's per-job
380         # try/except and marks just this one job errored.
381         scaler.step(optimizer)
382         scaler.update()
383         batch_losses.append(loss.item())
384
385     epoch_loss = float(np.mean(batch_losses)) if batch_losses else 0.0
386     epoch_losses.append(epoch_loss)
387     scheduler.step()
388
389     if (epoch + 1) % 10 == 0:
390         elapsed = (time.time() - t_start) / 60
391         print(f' [{diff_algo}/{scenario}/s{seed}] ',
392               f'epoch={epoch+1:3d} loss={epoch_loss:.6f} ',
393               f'elapsed={elapsed:.1f}min')
394
395     if epoch_loss < best_loss and epoch_loss > 0:
396         best_loss = epoch_loss
397         ckpt_path = os.path.join(
398             checkpoint_dir,
399             f'{diff_algo}_{rl_algo}_{scenario}_seed{seed}_best.pt')
400         torch.save(diff_model.state_dict(), ckpt_path)

```

```

401
402     if (epoch + 1) % 25 == 0 and torch.cuda.is_available():
403         torch.cuda.empty_cache()
404
405     final_path = os.path.join(
406         checkpoint_dir,
407         f'{diff_algo}_{rl_algo}_{scenario}_seed{seed}_final.pt')
408     torch.save(diff_model.state_dict(), final_path)
409
410     total_time = (time.time() - t_start) / 60
411     print(f'[{diff_algo}/{rl_algo}/{scenario}/seed={seed}] ',
412           f'Done in {total_time:.1f} min  best_loss={best_loss:.6f}')
413
414     return {
415         'diff_algo':      diff_algo,
416         'rl_algo':        rl_algo,
417         'scenario':        scenario,
418         'seed':           seed,
419         'diff_loss':       epoch_losses,
420         'best_loss':       best_loss,
421         'n_trajectories':  len(trajs),
422         'training_time_min': total_time,
423         'model_path':      final_path,
424     }

```

Listing B.18: Diffusion trainer with cosine noise schedule, DDPM noise prediction loss, smoothness regularisation and optional Min-SNR weighting.

```

1  """
2  training/statistical_eval.py
3  =====
4  Statistical analysis and structured Excel export for the 4-stage pipeline.
5

```

```

6 Excel workbook layout (one sheet per view):
7
8 rl_summary      --- 15 rows (5 RL algos × 3 scenarios): final
    completion %,
9
    best reward, wall-clock minutes, learning trend
    delta.
10 diff_summary    --- 15 rows (5 diff algos × 3 scenarios, all paired
    with
11
    HybridRL per DIFF_RL_PAIR): best/final loss,
    n_trajs.
12 hybrid_vs_base_rl --- 3 rows (one per scenario): HybridRL vs each
    base RL
13
    algo; makes the distillation gain directly
    inspectable.
14 hybrid_vs_base_diff --- 3 rows: HybridDiffusion vs each base diffusion
    model.
15 warehouse / sar / --- Per-scenario sheet combining RL and diff rows,
    matching
16 maritime_security thesis result tables.
17 all_results_raw  --- Every metric key, one row per job, for
    re-analysis.
18
19 StatisticalEvaluator static methods (Welch t, Mann-Whitney U, Cohen's d,
20 bootstrap CI) are available for paired multi-seed analyses when SEEDS is
21 widened beyond a single seed.
22 ""
23 from __future__ import annotations
24 import os
25 from typing import Dict, List, Any, Tuple, Optional
26
27 import numpy as np
28 import pandas as pd
29 from scipy import stats
30

```

```

31 from config.base_config import (
32     DIFF_ALGOS, DIFF_RL_PAIR, RL_ALGOS, SCENARIOS, SEEDS,
33     TASK_STATUS_AVAILABLE, TASK_STATUS_COMPLETE,
34 )
35
36
37 #
38 # Low-level statistical tests (used for multi-seed paired comparisons)
39 #
40
41 class StatisticalEvaluator:
42     """Welch t-test, Mann-Whitney U, Cohen's d, bootstrap CI."""
43
44     @staticmethod
45     def welch_t_test(a: np.ndarray, b: np.ndarray,
46                     alpha: float = 0.05) -> Dict:
47         if len(a) < 2 or len(b) < 2:
48             return {'t_stat': 0.0, 'p_value': 1.0, 'significant': False}
49         t, p = stats.ttest_ind(a, b, equal_var=False)
50         return {'t_stat': float(t), 'p_value': float(p),
51                 'significant': bool(p < alpha)}
52
53     @staticmethod
54     def mann_whitney_u(a: np.ndarray, b: np.ndarray,
55                       alpha: float = 0.05) -> Dict:
56         if len(a) < 3 or len(b) < 3:
57             return {'u_stat': 0.0, 'p_value': 1.0, 'significant': False}
58         u, p = stats.mannwhitneyu(a, b, alternative='two-sided')
59         return {'u_stat': float(u), 'p_value': float(p),
60                 'significant': bool(p < alpha)}
61
62     @staticmethod
63     def cohens_d(a: np.ndarray, b: np.ndarray) -> float:
64         if len(a) < 2 or len(b) < 2:

```

```

65         return 0.0
66         pooled_std = np.sqrt((np.var(a, ddof=1) + np.var(b, ddof=1)) / 2)
67         return float((np.mean(a) - np.mean(b)) / (pooled_std + 1e-9))
68
69     @staticmethod
70     def bootstrap_ci(x: np.ndarray, n_bootstrap: int = 2000,
71                     alpha: float = 0.05) -> Tuple[float, float]:
72         if len(x) < 2:
73             v = float(x[0]) if len(x) else 0.0
74             return v, v
75         rng = np.random.RandomState(42)
76         means = [rng.choice(x, len(x), replace=True).mean()
77                 for _ in range(n_bootstrap)]
78         lo = float(np.percentile(means, 100 * alpha / 2))
79         hi = float(np.percentile(means, 100 * (1 - alpha / 2)))
80         return lo, hi
81
82     @staticmethod
83     def descriptive(x: np.ndarray) -> Dict:
84         if len(x) == 0:
85             return {}
86         ci_lo, ci_hi = StatisticalEvaluator.bootstrap_ci(x)
87         return {
88             'n': len(x),
89             'mean': float(np.mean(x)),
90             'median': float(np.median(x)),
91             'std': float(np.std(x, ddof=1)) if len(x) > 1 else 0.0,
92             'sem': (float(np.std(x, ddof=1) / np.sqrt(len(x)))
93                    if len(x) > 1 else 0.0),
94             'min': float(np.min(x)),
95             'max': float(np.max(x)),
96             'q25': float(np.percentile(x, 25)),
97             'q75': float(np.percentile(x, 75)),
98             'ci_lo': ci_lo,

```

```

99         'ci_hi': ci_hi,
100     }
101
102
103 #
104 # Helpers for extracting per-job metrics from the results dict
105 #
106
107 def _get_rl(all_results: Dict, algo: str, scenario: str,
108             seed: int) -> Optional[Dict]:
109     return all_results.get(f'RL_{algo}_{scenario}_s{seed}')
110
111
112 def _get_diff(all_results: Dict, diff: str, rl: str, scenario: str,
113              seed: int) -> Optional[Dict]:
114     return all_results.get(f'DIFF_{diff}_{rl}_{scenario}_s{seed}')
115
116
117 def _rl_row(r: Optional[Dict], algo: str, scenario: str, seed: int) ->
    Dict:
118     """One row of the rl_summary table."""
119     out = {
120         'scenario':      scenario,
121         'rl_algo':      algo,
122         'seed':         seed,
123         'status':       r.get('status', 'missing') if r else
        'missing',
124         'final_completion': 0.0,
125         'best_reward':     0.0,
126         'avg_reward_last100': 0.0,
127         'std_reward_last100': 0.0,
128         'early_completion_pct': 0.0,
129         'late_completion_pct': 0.0,
130         'completion_delta_pp': 0.0,

```

```

131         'n_episodes':      0,
132         'training_min':    0.0,
133     }
134     if r is None or r.get('status') != 'ok':
135         return out
136     _raw_rewards = r.get('rewards')
137     rewards = _raw_rewards if isinstance(_raw_rewards, list) else []
138     _raw_comps = r.get('completion_rates')
139     comps = _raw_comps if isinstance(_raw_comps, list) else []
140     out['final_completion'] = float((r.get('final_completion', 0) or 0) *
141                                     100)
142     out['best_reward'] = float(r.get('best_reward', 0.0) or 0.0)
143     if rewards:
144         last = rewards[-100:]
145         out['avg_reward_last100'] = float(np.mean(last))
146         out['std_reward_last100'] = (float(np.std(last)) if len(last) > 1
147                                     else 0.0)
148     out['n_episodes'] = len(rewards)
149     out['training_min'] = float(r.get('training_time_min', 0.0) or 0.0)
150     if len(comps) >= 5:
151         n = max(1, len(comps) // 5)
152         early = float(np.mean(comps[:n])) * 100
153         late = float(np.mean(comps[-n:])) * 100
154         out['early_completion_pct'] = early
155         out['late_completion_pct'] = late
156         out['completion_delta_pp'] = late - early
157     return out
158
159 def _diff_row(r: Optional[Dict], diff: str, rl: str, scenario: str,
160             seed: int) -> Dict:
161     """One row of the diff_summary table."""
162     out = {
163         'scenario':      scenario,

```



```

163         'diff_algo':      diff,
164         'rl_algo':        rl,
165         'seed':           seed,
166         'status':         r.get('status', 'missing') if r else 'missing',
167         'best_loss':      float('nan'),
168         'final_loss':     float('nan'),
169         'early_loss':     float('nan'),
170         'loss_reduction': float('nan'),
171         'n_trajectories': 0,
172         'n_epochs_run':   0,
173     }
174     if r is None or r.get('status') != 'ok':
175         return out
176     bl = r.get('best_loss')
177     if isinstance(bl, (int, float)) and bl != float('inf'):
178         out['best_loss'] = float(bl)
179     losses = r.get('diff_loss') or r.get('epoch_losses') or []
180     if losses:
181         out['early_loss'] = float(losses[0])
182         out['final_loss'] = float(losses[-1])
183         if len(losses) >= 2 and losses[0] != 0:
184             out['loss_reduction'] = float(
185                 (losses[0] - losses[-1]) / abs(losses[0]))
186         out['n_epochs_run'] = len(losses)
187     out['n_trajectories'] = int(r.get('n_trajectories', 0) or 0)
188     return out
189
190
191 #
192 # Excel export
193 #
194
195 def export_results_to_excel(all_results: Dict,
196                             output_dir: str = 'results',

```

```

197         seed: Optional[int] = None) -> str:
198     """
199     Write the training_results.xlsx file with all sheets listed in the
200     module docstring. Caller passes the full results dict produced by
201     run.py's run_staged.
202     """
203     os.makedirs(output_dir, exist_ok=True)
204     path = os.path.join(output_dir, 'training_results.xlsx')
205     if seed is None:
206         seed = SEEDS[0]
207
208     # Build each sheet's DataFrame
209     rl_rows = [
210         _rl_row(_get_rl(all_results, a, s, seed), a, s, seed)
211         for a in RL_ALGOS for s in SCENARIOS
212     ]
213     df_rl = pd.DataFrame(rl_rows)
214
215     diff_rows = [
216         _diff_row(_get_diff(all_results, d, DIFF_RL_PAIR, s, seed),
217                 d, DIFF_RL_PAIR, s, seed)
218         for d in DIFF_ALGOS for s in SCENARIOS
219     ]
220     df_diff = pd.DataFrame(diff_rows)
221
222     # HybridRL vs base-RL comparison (wide format, one row per scenario)
223     base_rl = [a for a in RL_ALGOS if a != 'HybridRL']
224     hyb_rl_rows = []
225     for s in SCENARIOS:
226         row: Dict[str, Any] = {'scenario': s}
227         hyb = _get_rl(all_results, 'HybridRL', s, seed)
228         row['HybridRL_completion_pct'] = (
229             float((hyb.get('final_completion', 0) or 0) * 100)
230             if hyb and hyb.get('status') == 'ok' else float('nan'))

```

```

231     for a in base_rl:
232         br = _get_rl(all_results, a, s, seed)
233         row[f'{a}_completion_pct'] = (
234             float((br.get('final_completion', 0) or 0) * 100)
235             if br and br.get('status') == 'ok' else float('nan'))
236         # Best base as benchmark
237         base_vals = [row[f'{a}_completion_pct'] for a in base_rl
238                     if not np.isnan(row[f'{a}_completion_pct'])]
239         row['best_base_completion_pct'] = (
240             float(max(base_vals)) if base_vals else float('nan'))
241         row['hybrid_advantage_pp'] = (
242             row['HybridRL_completion_pct'] -
row['best_base_completion_pct']
243             if not (np.isnan(row['HybridRL_completion_pct']) or
244                     np.isnan(row['best_base_completion_pct']))
245             else float('nan'))
246         hyb_rl_rows.append(row)
247     df_hyb_rl = pd.DataFrame(hyb_rl_rows)
248
249     # HybridDiffusion vs base-diff comparison
250     base_diff = [d for d in DIFF_ALGOS if d != 'HybridDiffusion']
251     hyb_diff_rows = []
252     for s in SCENARIOS:
253         row = {'scenario': s}
254         hyb = _get_diff(all_results, 'HybridDiffusion', DIFF_RL_PAIR, s,
seed)
255         row['HybridDiffusion_best_loss'] = (
256             float(hyb.get('best_loss', float('nan'))))
257             if hyb and hyb.get('status') == 'ok' else float('nan'))
258         for d in base_diff:
259             br = _get_diff(all_results, d, DIFF_RL_PAIR, s, seed)
260             row[f'{d}_best_loss'] = (
261                 float(br.get('best_loss', float('nan'))))
262                 if br and br.get('status') == 'ok' else float('nan'))

```

```

263     base_vals = [row[f'{d}_best_loss'] for d in base_diff
264                  if not np.isnan(row[f'{d}_best_loss'])]
265     # Lower loss is better, so the base benchmark is the minimum
266     row['best_base_loss'] = (
267         float(min(base_vals)) if base_vals else float('nan'))
268     row['hybrid_loss_reduction'] = (
269         (row['best_base_loss'] - row['HybridDiffusion_best_loss'])
270         if not (np.isnan(row['HybridDiffusion_best_loss']) or
271               np.isnan(row['best_base_loss']))
272         else float('nan'))
273     hyb_diff_rows.append(row)
274 df_hyb_diff = pd.DataFrame(hyb_diff_rows)
275
276 # Per-scenario combined sheet
277 per_scenario_frames: Dict[str, pd.DataFrame] = {}
278 for s in SCENARIOS:
279     rl_part = df_rl[df_rl['scenario'] == s].copy()
280     rl_part.insert(0, 'kind', 'RL')
281     # Unify the column sets for vertical stacking
282     rl_part = rl_part[[
283         'kind', 'scenario', 'rl_algo', 'status',
284         'final_completion', 'best_reward', 'completion_delta_pp',
285         'n_episodes', 'training_min',
286     ]].rename(columns={'rl_algo': 'algo',
287                       'final_completion': 'metric_pct_or_loss',
288                       'best_reward': 'best_reward_or_loss'})
289     diff_part = df_diff[df_diff['scenario'] == s].copy()
290     diff_part.insert(0, 'kind', 'Diff')
291     diff_part = diff_part[[
292         'kind', 'scenario', 'diff_algo', 'status',
293         'final_loss', 'best_loss', 'loss_reduction',
294         'n_epochs_run', 'n_trajectories',
295     ]].rename(columns={'diff_algo': 'algo',
296                       'final_loss': 'metric_pct_or_loss',

```

```

297         'best_loss': 'best_reward_or_loss',
298         'loss_reduction': 'completion_delta_pp',
299         'n_epochs_run': 'n_episodes',
300         'n_trajectories': 'training_min'})
301     per_scenario_frames[s] = pd.concat([rl_part, diff_part],
ignore_index=True)
302
303     # Raw results table - one row per job key, useful for re-analysis
304     raw_rows = []
305     for key, res in all_results.items():
306         rewards = res.get('rewards', [])
307         comp     = res.get('completion_rates', [])
308         raw_rows.append({
309             'job_key':         key,
310             'scenario':        res.get('scenario', ''),
311             'rl_algo':         res.get('algo', ''),
312             'diff_algo':       res.get('diff_algo', ''),
313             'seed':            res.get('seed', 0),
314             'status':          res.get('status', ''),
315             'avg_reward':      (float(np.mean(rewards[-100:]))) if rewards
else 0.0),
316             'std_reward':      (float(np.std(rewards[-100:])))
if len(rewards) > 1 else 0.0),
317             'task_completion': (float(np.mean(comp[-100:]))) if comp else
0.0),
318             'best_reward':     res.get('best_reward', 0.0),
319             'training_min':    res.get('training_time_min', 0.0),
320             'diff_best_loss':  res.get('best_loss', None),
321             'n_trajs':         res.get('n_trajectories', None),
322             'error':           str(res.get('error', '')[0:200])
if res.get('status') != 'ok' else '',
323         })
324     df_raw = pd.DataFrame(raw_rows)
325
326
327

```

```

328     # Write everything
329     with pd.ExcelWriter(path, engine='openpyxl') as writer:
330         df_rl.to_excel(writer,          sheet_name='rl_summary',
331                             index=False)
332         df_diff.to_excel(writer,        sheet_name='diff_summary',
333                             index=False)
334         df_hyb_rl.to_excel(writer,      sheet_name='hybrid_vs_base_rl',
335                             index=False)
336         df_hyb_diff.to_excel(writer,    sheet_name='hybrid_vs_base_diff',
337                             index=False)
338         for s, df in per_scenario_frames.items():
339             df.to_excel(writer, sheet_name=s[:31], index=False)
340             df_raw.to_excel(writer, sheet_name='all_results_raw', index=False)
341
342     print(f'Results exported -> {path}')
343     print(f'  rl_summary          : {len(df_rl)} rows ({len(RL_ALGOS)}
344             algos x {len(SCENARIOS)} scenarios)')
345     print(f'  diff_summary       : {len(df_diff)} rows
346             ({len(DIFF_ALGOS)} diff on {DIFF_RL_PAIR} x {len(SCENARIOS)}
347             scenarios)')
348     print(f'  hybrid_vs_base_rl   : {len(df_hyb_rl)} rows')
349     print(f'  hybrid_vs_base_diff : {len(df_hyb_diff)} rows')
350     print(f'  all_results_raw    : {len(df_raw)} rows')
351     return path
352
353 #
354 # Greedy FIFO baseline (TA-RWARE heuristic) - kept from as a comparison
355     point.
356 # note: no fallback try/except wrappers inside the rollout loop - if the
357 # env raises during baseline eval we surface that error rather than
358     silently
359 # producing degenerate metrics.
360 #

```

```

353
354 def run_greedy_fifo_baseline(env, n_episodes: int = 100,
355                             max_steps: int = 200, seed: int = 42) ->
    dict:
356     """
357     TA-RWARE FIFO greedy baseline: nearest robot -> nearest pending task.
358     Euclidean distance only (no A*).
359     """
360     completions, makespans, travel_dists = [], [], []
361
362     for ep in range(n_episodes):
363         obs = env.reset(seed=seed + ep)
364         if isinstance(obs, tuple):
365             obs = obs[0]
366
367         step = 0
368         done = False
369         ep_travel = 0.0
370         _prev_positions: Dict[int, Tuple[float, float]] = {}
371
372         while not done and step < max_steps:
373             robots = getattr(env, 'robots', [])
374             tasks = getattr(env, 'tasks', [])
375             assignments = getattr(env, 'task_assignments', {})
376             completed = getattr(env, 'completed_tasks', set())
377
378             pending = [t for t in tasks
379                        if t.get('id') not in assignments
380                           and t.get('id') not in completed
381                           and t.get('status', TASK_STATUS_AVAILABLE) ==
TASK_STATUS_AVAILABLE]
382             free_robots = [r for r in robots
383                           if r.get('id') not in assignments.values()]
384

```

```

385         action = 0    # no-op by default
386     if pending and free_robots:
387         best_dist = float('inf')
388         best_task_id = 0
389         for r in free_robots:
390             rpos = np.array(r.get('position', [0., 0.]))
391             for t in pending:
392                 tloc = np.array(t.get('location', [0., 0.]))
393                 d = np.linalg.norm(rpos - tloc)
394                 if d < best_dist:
395                     best_dist = d
396                     best_task_id = t.get('id', 0)
397             action = best_task_id
398
399     obs, reward, term, trunc, info = env.step(action)
400     done = term or trunc
401
402     # Track travel distance
403     for r in robots:
404         rid = r.get('id', 0)
405         rpos = tuple(r.get('position', [0., 0.]))
406         if rid in _prev_positions:
407             ep_travel += float(np.linalg.norm(
408                 np.array(rpos) - np.array(_prev_positions[rid])))
409             _prev_positions[rid] = rpos
410
411     step += 1
412
413     tasks = getattr(env, 'tasks', [])
414     completed = getattr(env, 'completed_tasks', set())
415     n_done = (len(completed) if completed
416              else sum(1 for t in tasks
417                      if t.get('status') == TASK_STATUS_COMPLETE))
418     n_total = len(tasks) if tasks else 1

```



```

419
420     completions.append(n_done / max(n_total, 1))
421     makespans.append(step)
422     travel_dists.append(ep_travel)
423
424     return {
425         'baseline':      'greedy_fifo',
426         'completion_rate': (float(np.mean(completions))
427                               if completions else 0.0),
428         'completion_std': (float(np.std(completions))
429                               if completions else 0.0),
430         'makespan':      (float(np.mean(makespans))
431                               if makespans else float(max_steps)),
432         'travel_dist':   (float(np.mean(travel_dists))
433                               if travel_dists else 0.0),
434         'n_episodes':    n_episodes,
435     }

```

Listing B.19: Statistical evaluation: Welch’s t-test, Mann-Whitney U test, Cohen’s d effect size, bootstrap confidence intervals and the greedy first-in-first-out baseline.

```

1  """
2  training/visualizer.py
3  =====
4  Comprehensive visualization engine for AQUA Guardian.
5
6  Generates publication-ready PNG figures for thesis and paper.
7
8  Run after training:
9      from training.visualizer import generate_all_figures
10     generate_all_figures('results/', 'results/figures/')
11
12 Output structure:

```

```

13  figures/
14  eda/ Environment analysis (run before training)
15  01_robot_params.png
16  02_task_heatmaps.png
17  03_scenario_complexity.png
18  04_population_distributions.png
19  rl/ RL training results
20  05_learning_curves.png
21  06_completion_rates.png
22  07_algorithm_comparison.png
23  08_makespan_comparison.png
24  09_task_allocation_heatmap.png
25  10_task_dependency_graph.png
26  11_hybrid_bandit_selection.png
27  12_variable_robot_performance.png
28  diffusion/ Diffusion model results
29  13_diffusion_loss_curves.png
30  14_trajectory_visualization.png
31  15_bspline_vs_waypoints.png
32  16_denoising_sequence.png
33  17_inter_robot_avoidance.png
34  18_goal_conditioning.png
35  combined/ Combined comparison figures
36  19_rl_diff_combined.png
37  20_summary_table.png
38  """
39
40  import os
41  import pickle
42  import numpy as np
43  import matplotlib
44  matplotlib.use('Agg') # headless - no display needed on server
45  import matplotlib.pyplot as plt
46  import matplotlib.patches as mpatches

```

```

47 import matplotlib.gridspec as gridspec
48 from matplotlib.colors import LinearSegmentedColormap
49 from typing import Dict, Any, List, Optional, Tuple
50 from pathlib import Path
51
52 # Plot style
53 DARK_BG      = '#0d1117'
54 PANEL_BG     = '#161b22'
55 GRID_COLOR   = '#21262d'
56 TEXT_COLOR   = '#c9d1d9'
57 ACCENT       = '#58a6ff'
58
59 ALGO_COLORS = {
60     'MAPPO':      '#2196F3',
61     'Q MIX':      '#4CAF50',
62     'MADDPG':     '#FF9800',
63     'DQN':        '#E91E63',
64     'HybridRL':   '#FFEB3B',
65 }
66 DIFF_COLORS = {
67     'TDM':         '#00BCD4',
68     'DiffusionPolicy': '#9C27B0',
69     'Diffuser':     '#FF5722',
70     'SDE':          '#8BC34A',
71     'HybridDiffusion': '#FFC107',
72 }
73 SCENARIO_COLORS = {
74     'warehouse':    '#2196F3',
75     'search_and_rescue': '#FF5722',
76     'maritime_security': '#4CAF50',
77 }
78 SCENARIO_LABELS = {
79     'warehouse':    'Warehouse\n(20×20 m)',
80     'search_and_rescue': 'Search & Rescue\n(50×50 m)',

```

```

81     'maritime_security': 'Maritime Security\n(200×200 m)',
82 }
83
84 def _setup_dark_style():
85     plt.rcParams.update({
86         'figure.facecolor': DARK_BG,
87         'axes.facecolor': PANEL_BG,
88         'axes.edgecolor': GRID_COLOR,
89         'axes.labelcolor': TEXT_COLOR,
90         'axes.titlecolor': TEXT_COLOR,
91         'xtick.color': TEXT_COLOR,
92         'ytick.color': TEXT_COLOR,
93         'grid.color': GRID_COLOR,
94         'text.color': TEXT_COLOR,
95         'legend.facecolor': PANEL_BG,
96         'legend.edgecolor': GRID_COLOR,
97         'legend.labelcolor': TEXT_COLOR,
98         'font.size': 10,
99         'axes.titlesize': 12,
100        'axes.titleweight': 'bold',
101        'savefig.facecolor': DARK_BG,
102        'savefig.dpi': 150,
103        'savefig.bbox': 'tight',
104    })
105
106 def _smooth(x: np.ndarray, w: int = 50) -> np.ndarray:
107     """Smooth a 1-D array with a uniform window."""
108     if len(x) < w:
109         return x
110     return np.convolve(x, np.ones(w) / w, mode='valid')
111
112 def _save(fig, path: str):
113     os.makedirs(os.path.dirname(path), exist_ok=True)
114     fig.savefig(path)

```

```

115     plt.close(fig)
116     print(f'    Saved → {path}')
```

117

118

```

119     #
120     # EDA FIGURES
121     #
122
```

```

123 def plot_robot_params(out_dir: str):
124     """
125     Figure 01 - Robot parameters: ONE ROW PER SCENARIO, large clear subplots.
126     Each row shows all parameters for that scenario with value annotations.
127     """
128     from config.base_config import SCENARIO_CONFIGS
129     _setup_dark_style()
130
131     scenarios = ['warehouse', 'search_and_rescue', 'maritime_security']
132     sc_names  = ['Warehouse (20×20 m)', 'Search & Rescue (50×50 m)',
133                 'Maritime Security (200×200 m)']
134     params    = ['max_speed', 'battery_capacity', 'max_payload',
135                 'observation_radius', 'communication_range',
136                 'battery_drain_rate']
137     labels    = ['Max Speed\n(m/s)', 'Battery\n(units)', 'Max
138                 Payload\n(kg)',
139                 'Obs Radius\n(m)', 'Comm Range\n(m)', 'Drain
140                 Rate\n(/step)']
141
142     fig, axes = plt.subplots(3, len(params), figsize=(18, 12))
143     fig.suptitle('Robot Parameters --- Per Scenario Breakdown',
144                 fontsize=14, fontweight='bold')
145
146     for row, (sc, sc_name) in enumerate(zip(scenarios, sc_names)):
147         cfg = SCENARIO_CONFIGS[sc]
148         col_base = SCENARIO_COLORS[sc]
```

```

145         for col, (param, label) in enumerate(zip(params, labels)):
146             ax = axes[row, col]
147             val = getattr(cfg, param, 0)
148             bar = ax.bar([0], [val], color=col_base, alpha=0.85,
149                           edgecolor='white', linewidth=0.8, width=0.6)
150             ax.text(0, val * 1.03, f'{val:.3g}',
151                     ha='center', va='bottom', fontsize=10,
152                     fontweight='bold', color='white')
153             if col == 0:
154                 ax.set_ylabel(sc_name, fontsize=9, color=col_base,
155                               fontweight='bold')
156                 if row == 0:
157                     ax.set_title(label, fontsize=9, fontweight='bold')
158                     ax.set_xticks([])
159                     ax.grid(axis='y', alpha=0.3)
160                     # Add raw value as subtitle
161                     ax.set_xlabel(f'{val:.3g}', fontsize=8, color=TEXT_COLOR,
162                                   labelpad=1)
163
164     plt.tight_layout(rect=[0, 0, 1, 0.96])
165     _save(fig, f'{out_dir}/eda/01_robot_params.png')
166
167 def plot_task_location_heatmaps(out_dir: str):
168     """
169     Figure 02 - Task location density heatmaps, ONE FIGURE PER SCENARIO
170     saved separately.
171     Also saves a combined 1x3 figure.
172     Simulates realistic task spawning patterns for each scenario type.
173     """
174     from config.base_config import SCENARIO_CONFIGS
175     _setup_dark_style()
176     np.random.seed(42)

```

```

175 fig_all, axes_all = plt.subplots(1, 3, figsize=(18, 6))
176 fig_all.suptitle('Task Location Density Maps --- All Scenarios',
177                 fontsize=13, fontweight='bold')
178
179 scenarios = ['warehouse', 'search_and_rescue', 'maritime_security']
180 task_configs = {
181     'warehouse': dict(n=500, pattern='grid', noise=0.08),
182     'search_and_rescue': dict(n=500, pattern='clusters', noise=0.25),
183     'maritime_security': dict(n=200, pattern='uniform', noise=0.4),
184 }
185
186 for idx, sc in enumerate(scenarios):
187     cfg = SCENARIO_CONFIGS[sc]
188     gh = cfg.grid_half
189     tc = task_configs[sc]
190     col = SCENARIO_COLORS[sc]
191
192     # Generate task locations by pattern
193     if tc['pattern'] == 'grid':
194         # Warehouse: tasks shelf positions (grid pattern)
195         gx, gy = np.meshgrid(np.linspace(-gh*0.7, gh*0.7, 8),
196                             np.linspace(-gh*0.7, gh*0.7, 6))
197         base_x = gx.ravel() + np.random.randn(48) * gh * tc['noise']
198         base_y = gy.ravel() + np.random.randn(48) * gh * tc['noise']
199         # Repeat to get 500
200         reps = tc['n'] // 48 + 1
201         task_x = np.tile(base_x, reps)[:tc['n']] +
202             np.random.randn(tc['n']) * gh * 0.02
203         task_y = np.tile(base_y, reps)[:tc['n']] +
204             np.random.randn(tc['n']) * gh * 0.02
205
206     elif tc['pattern'] == 'clusters':
207         # SAR: task victims in 4-6 clustered hotspots
208         n_clusters = 5

```

```

206         centres      = np.random.uniform(-gh*0.7, gh*0.7, (n_clusters,
207         2))
208         task_x, task_y = [], []
209         for ci in range(n_clusters):
210             n_here = tc['n'] // n_clusters
211             task_x.extend(centres[ci, 0] + np.random.randn(n_here) *
212             gh * tc['noise'])
213             task_y.extend(centres[ci, 1] + np.random.randn(n_here) *
214             gh * tc['noise'])
215             task_x = np.array(task_x); task_y = np.array(task_y)
216
217         else:
218             # Maritime: sparse uniform distribution
219             task_x = np.random.uniform(-gh*0.85, gh*0.85, tc['n'])
220             task_y = np.random.uniform(-gh*0.85, gh*0.85, tc['n'])
221
222             task_x = np.clip(task_x, -gh, gh)
223             task_y = np.clip(task_y, -gh, gh)
224
225             # Per-scenario individual figure
226             fig_sc, ax_sc = plt.subplots(figsize=(7, 7))
227             fig_sc.patch.set_facecolor(DARK_BG)
228             ax_sc.set_facecolor('#060a10')
229             ax_sc.set_xlim(-gh, gh); ax_sc.set_ylim(-gh, gh)
230             ax_sc.set_aspect('equal')
231             sc_full = sc.replace('_', ' ').title()
232             ax_sc.set_title(f'Task Location Density ---
233             {sc_full}\n({gh*2:.0f}×{gh*2:.0f} m map)',
234             fontsize=12, fontweight='bold', color=TEXT_COLOR,
235             pad=12)
236             ax_sc.set_xlabel('X Position (m)', color=TEXT_COLOR)
237             ax_sc.set_ylabel('Y Position (m)', color=TEXT_COLOR)
238
239             # KDE-style density via hexbin

```



```

235     hb = ax_sc.hexbin(task_x, task_y, gridsize=20, cmap='Blues',
236                       mincnt=1, extent=[-gh, gh, -gh, gh])
237     cb = fig_sc.colorbar(hb, ax=ax_sc, label='Task frequency')
238     cb.ax.yaxis.set_tick_params(color=TEXT_COLOR)
239
240     # Overlay scatter
241     ax_sc.scatter(task_x[:,5], task_y[:,5], s=6, c=col,
242                  alpha=0.25, zorder=3, label='Task locations')
243
244     # Draw map boundary
245     rect = plt.Rectangle((-gh, -gh), gh*2, gh*2,
246                           fill=False, edgecolor=col, linewidth=1.5,
247                           linestyle='--')
248     ax_sc.add_patch(rect)
249
250     ax_sc.grid(color=GRID_COLOR, alpha=0.3)
251     ax_sc.legend(loc='upper right', fontsize=9)
252
253     safe_sc = sc.replace('_', '')[3:].upper()
254     _save(fig_sc, f'{out_dir}/eda/02_task_heatmap_{sc}.png')
255
256     # Combined subplot
257     ax = axes_all[idx]
258     ax.set_facecolor('#060a10')
259     ax.set_xlim(-gh, gh); ax.set_ylim(-gh, gh)
260     ax.set_aspect('equal')
261     ax.set_title(f'{sc_full}\n({gh*2:.0f}×{gh*2:.0f} m)',
262                  fontsize=10, fontweight='bold')
263     ax.set_xlabel('X (m)'); ax.set_ylabel('Y (m)')
264     ax.hexbin(task_x, task_y, gridsize=18, cmap='Blues',
265               mincnt=1, extent=[-gh, gh, -gh, gh])
266     ax.scatter(task_x[:,5], task_y[:,5], s=4, c=col, alpha=0.2,
267               zorder=3)
268     ax.grid(color=GRID_COLOR, alpha=0.3)

```

```

266
267     plt.tight_layout()
268     _save(fig_all, f'{out_dir}/eda/02_task_heatmaps_combined.png')
269
270
271 def plot_scenario_complexity(out_dir: str):
272     """
273     Figure 03 - Scenario complexity: heatmap + per-scenario radar charts
274     side by side.
275     Clear visual for thesis comparison chapter.
276     """
277     from config.base_config import SCENARIO_CONFIGS
278     _setup_dark_style()
279
280     scenarios = ['warehouse', 'search_and_rescue', 'maritime_security']
281     sc_labels = ['Warehouse', 'Search & Rescue', 'Maritime']
282     metrics    = ['grid_half', 'max_tasks', 'num_robots', 'max_speed',
283                  'observation_radius', 'battery_capacity']
284     m_labels   = ['Map Size', 'Tasks', 'Robots', 'Speed', 'Obs Range',
285                  'Battery']
286
287     data = np.array([[getattr(SCENARIO_CONFIGS[sc], m) for m in metrics]
288                     for sc in scenarios], dtype=float)
289
290     col_min = data.min(axis=0); col_max = data.max(axis=0)
291     norm     = (data - col_min) / (col_max - col_min + 1e-9)
292
293     fig = plt.figure(figsize=(18, 7))
294     fig.suptitle('Scenario Complexity Analysis', fontsize=14,
295                 fontweight='bold')
296
297     # Left: heatmap (half width)
298     ax_heat = fig.add_subplot(1, 4, (1, 2))
299     cmap     = LinearSegmentedColormap.from_list('aq', [PANEL_BG,
300                 '#1a3a6b', ACCENT, '#ffffff'])

```

```

296     im      = ax_heat.imshow(norm, cmap=cmap, aspect='auto', vmin=0,
297                               vmax=1)
298     ax_heat.set_xticks(range(len(m_labels)))
299     ax_heat.set_xticklabels(m_labels, rotation=35, ha='right',
300                             fontsize=10)
301     ax_heat.set_yticks(range(3))
302     ax_heat.set_yticklabels(sc_labels, fontsize=11, fontweight='bold')
303     for i in range(3):
304         for j in range(len(metrics)):
305             raw_val = data[i, j]
306             txt_col = 'white' if norm[i,j] < 0.55 else '#0d1117'
307             ax_heat.text(j, i, f'{raw_val:.0f}\n({norm[i,j]:.2f})',
308                           ha='center', va='center', fontsize=8,
309                           color=txt_col)
310     fig.colorbar(im, ax=ax_heat, shrink=0.8, label='Normalised
311     Complexity')
312     ax_heat.set_title('Complexity Heatmap\n(raw value / normalised)',
313                       fontsize=11, fontweight='bold')
314
315     # Right: one radar per scenario
316     angles = np.linspace(0, 2*np.pi, len(m_labels),
317                           endpoint=False).tolist()
318     angles += angles[:1]
319
320     for ri, (sc, sc_name) in enumerate(zip(scenarios, sc_labels)):
321         ax_r = fig.add_subplot(1, 4, ri + 2, projection='polar')
322         ax_r.set_facecolor(PANEL_BG)
323         ax_r.set_title(sc_name, fontsize=10, fontweight='bold', pad=12,
324                         color=SCENARIO_COLORS[sc])
325         ax_r.set_xticks(angles[:-1])
326         ax_r.set_xticklabels(m_labels, fontsize=8)
327         ax_r.set_yticklabels([])
328         ax_r.grid(color=GRID_COLOR, alpha=0.5)
329         ax_r.set_ylim(0, 1)

```

```

324
325     v = list(norm[ri]) + [norm[ri][0]]
326     ax_r.plot(angles, v, 'o-', color=SCENARIO_COLORS[sc],
327               linewidth=2, markersize=5)
328
329     # Annotate each vertex with raw value
330     for ai, (ang, val, raw) in enumerate(zip(angles[:-1], norm[ri],
331 data[ri])):
332         ax_r.annotate(f'{raw:.0f}',
333                       xy=(ang, val), fontsize=7,
334                       color=SCENARIO_COLORS[sc], ha='center')
335
336     plt.tight_layout(rect=[0, 0, 1, 0.94])
337     _save(fig, f'{out_dir}/eda/03_scenario_complexity.png')
338
339 def plot_population_distributions(out_dir: str):
340     """
341     Figure 04 - Population distributions: SEPARATE FIGURE PER SCENARIO
342     + combined overview. Each scenario gets its own 1×4 panel with:
343     task priorities, payload, deadline, battery capacity distributions.
344     """
345     from config.base_config import SCENARIO_CONFIGS, SCENARIOS
346     _setup_dark_style()
347     np.random.seed(42)
348
349     scenarios = ['warehouse', 'search_and_rescue', 'maritime_security']
350
351     # One figure per scenario
352     for sc in scenarios:
353         cfg      = SCENARIO_CONFIGS[sc]
354         col      = SCENARIO_COLORS[sc]
355         sc_name  = sc.replace('_', ' ').title()

```

```

356
357     # Generate representative distributions
358     n = 300
359     priorities = np.random.choice([1,2,3,4,5], n,
360                                   p=[0.1,0.2,0.4,0.2,0.1])
361     payloads   = np.random.uniform(0.1, cfg.max_task_payload, n)
362     deadlines  = np.random.uniform(cfg.max_task_payload*0.5,
363                                   cfg.max_deadline, n)
364     battery    = np.random.normal(cfg.battery_capacity,
365                                   cfg.battery_capacity * 0.1, n)
366     battery    = np.clip(battery, 0, cfg.battery_capacity * 1.2)
367
368     fig, axes = plt.subplots(1, 4, figsize=(16, 5))
369     fig.suptitle(f'Population Distributions --- {sc_name}',
370                 fontsize=13, fontweight='bold', color=col)
371
372     # Task priority
373     ax = axes[0]
374     counts = [np.sum(priorities == p) for p in range(1, 6)]
375     ax.bar(range(1, 6), counts, color=col, alpha=0.85,
376           edgecolor='white', linewidth=0.5)
377     for i, c in enumerate(counts):
378         ax.text(i+1, c+2, str(c), ha='center', va='bottom',
379               fontsize=9)
380
381     ax.set_title('Task Priority Distribution', fontweight='bold')
382     ax.set_xlabel('Priority Level (1=Low, 5=High)')
383     ax.set_ylabel('Number of Tasks')
384     ax.set_xticks(range(1, 6))
385     ax.grid(axis='y', alpha=0.3)
386
387     # Payload
388     ax = axes[1]
389     ax.hist(payloads, bins=20, color=col, alpha=0.85,
390           edgecolor='white', linewidth=0.3)

```

```

387     ax.axvline(np.mean(payloads), color='white', linestyle='--',
388                linewidth=1.2, label=f'Mean: {np.mean(payloads):.0f}')
389     ax.set_title('Task Payload Distribution', fontweight='bold')
390     ax.set_xlabel(f'Payload (kg) --- max={cfg.max_task_payload:.0f}')
391     ax.set_ylabel('Count')
392     ax.legend(fontsize=8)
393     ax.grid(alpha=0.3)
394
395     # Deadline
396     ax = axes[2]
397     ax.hist(deadlines, bins=20, color=col, alpha=0.85,
398            edgecolor='white', linewidth=0.3)
399     ax.axvline(np.mean(deadlines), color='white', linestyle='--',
400                linewidth=1.2, label=f'Mean: {np.mean(deadlines):.0f}')
401     ax.set_title('Task Deadline Distribution', fontweight='bold')
402     ax.set_xlabel('Deadline (steps)')
403     ax.set_ylabel('Count')
404     ax.legend(fontsize=8)
405     ax.grid(alpha=0.3)
406
407     # Robot battery
408     ax = axes[3]
409     ax.hist(battery, bins=20, color=col, alpha=0.85,
410            edgecolor='white', linewidth=0.3)
411     ax.axvline(cfg.battery_capacity, color='#FFEB3B', linestyle='-.',
412                linewidth=1.5, label=f'Nominal:
413 {cfg.battery_capacity:.0f}')
414     ax.set_title('Robot Battery Capacity', fontweight='bold')
415     ax.set_xlabel('Battery (units)')
416     ax.set_ylabel('Count')
417     ax.legend(fontsize=8)
418     ax.grid(alpha=0.3)
419
420     plt.tight_layout(rect=[0, 0, 1, 0.93])

```

```

418         _save(fig, f'{out_dir}/eda/04_distributions_{sc}.png')
419
420     # Combined overview (2x3)
421     fig, axes = plt.subplots(2, 3, figsize=(16, 9))
422     fig.suptitle('Population Distributions --- All Scenarios Overview',
423                 fontweight='bold')
424     np.random.seed(42)
425
426     for col_idx, sc in enumerate(scenarios):
427         cfg = SCENARIO_CONFIGS[sc]
428         col = SCENARIO_COLORS[sc]
429         n = 300
430         priorities = np.random.choice([1,2,3,4,5], n,
431                                     p=[0.1,0.2,0.4,0.2,0.1])
432         payloads = np.random.uniform(0.1, cfg.max_task_payload, n)
433
434         ax_p = axes[0, col_idx]
435         ax_l = axes[1, col_idx]
436         sc_title = sc.replace('_', ' ').title()
437
438         counts = [np.sum(priorities == p) for p in range(1, 6)]
439         ax_p.bar(range(1,6), counts, color=col, alpha=0.85,
440                 edgecolor='white', linewidth=0.3)
441         ax_p.set_title(f'{sc_title}\nTask Priority', fontsize=10,
442                       fontweight='bold')
443         ax_p.set_xlabel('Priority'); ax_p.set_ylabel('Count')
444         ax_p.set_xticks(range(1,6)); ax_p.grid(axis='y', alpha=0.3)
445
446         ax_l.hist(payloads, bins=15, color=col, alpha=0.85,
447                 edgecolor='white', linewidth=0.3)
448         ax_l.axvline(np.mean(payloads), color='white', linestyle='--',
449                     linewidth=1)
450         ax_l.set_title(f'Payload (max={cfg.max_task_payload:.0f} kg)',
451                       fontsize=9)

```

```

445         ax_1.set_xlabel('Payload (kg)'); ax_1.set_ylabel('Count')
446         ax_1.grid(alpha=0.3)
447
448     plt.tight_layout()
449     _save(fig, f'{out_dir}/eda/04_distributions_combined.png')
450
451
452 def plot_map_layouts(out_dir: str):
453     """
454     Figure 00 - Environment map layouts: ONE FIGURE PER SCENARIO showing
455     the full map with task zones, robot spawn areas, obstacles, and
456     boundaries.
457     Gives a clear spatial understanding for the thesis.
458     """
459     from config.base_config import SCENARIO_CONFIGS
460     _setup_dark_style()
461     np.random.seed(42)
462
463     scenarios = ['warehouse', 'search_and_rescue', 'maritime_security']
464     obstacle_configs = {
465         'warehouse': dict(n=12, size_range=(0.3, 0.8),
466                          shelf_rows=True),
467         'search_and_rescue': dict(n=6, size_range=(1.5, 4.0),
468                                  shelf_rows=False),
469         'maritime_security': dict(n=3, size_range=(5.0, 15.0),
470                                  shelf_rows=False),
471     }
472
473     fig_all, axes_all = plt.subplots(1, 3, figsize=(18, 7))
474     fig_all.suptitle('Environment Map Layouts --- All Scenarios',
475                     fontsize=13, fontweight='bold')
476
477     for idx, sc in enumerate(scenarios):
478         cfg = SCENARIO_CONFIGS[sc]

```



```

474         gh          = cfg.grid_half
475         oc          = obstacle_configs[sc]
476         col         = SCENARIO_COLORS[sc]
477         sc_name      = sc.replace('_', ' ').title()
478
479         for ax, is_individual in [(axes_all[idx], False)] + \
480                                   [(plt.subplots(figsize=(8,8))[1],
True)]:
481             if is_individual:
482                 fig_sc = ax.figure
483                 fig_sc.patch.set_facecolor(DARK_BG)
484
485                 ax.set_facecolor('#060c18')
486                 ax.set_xlim(-gh*1.05, gh*1.05); ax.set_ylim(-gh*1.05, gh*1.05)
487                 ax.set_aspect('equal')
488                 title_str = f'{sc_name}\n{int(gh*2)}×{int(gh*2)} m'
489                 ax.set_title(title_str, fontsize=11 if is_individual else 10,
fontweight='bold')
490                 ax.set_xlabel('X Position (m)'); ax.set_ylabel('Y Position
(m)')
491
492                 # Map boundary
493                 boundary = plt.Rectangle((-gh, -gh), gh*2, gh*2,
494                                         fill=False, edgecolor=col,
495                                         linewidth=2, linestyle='-',
zorder=5)
496                 ax.add_patch(boundary)
497
498                 # Shelf rows (warehouse only)
499                 if oc['shelf_rows']:
500                     for row_y in np.linspace(-gh*0.7, gh*0.7, 5):
501                         for col_x in np.linspace(-gh*0.7, gh*0.7, 6):
502                             shelf = plt.Rectangle((col_x - gh*0.04, row_y -
gh*0.04),

```

```

503                                     gh*0.08, gh*0.08,
504                                     color='#455A64',
alpha=0.7, zorder=2)
505                                     ax.add_patch(shelf)
506
507     # Obstacles
508     for _ in range(oc['n']):
509         pos = np.random.uniform(-gh*0.75, gh*0.75, 2)
510         size = np.random.uniform(*oc['size_range'])
511         circ = plt.Circle(pos, size, color='#B71C1C', alpha=0.55,
zorder=3)
512         ax.add_patch(circ)
513         ax.plot(*pos, 'x', color='#FF5252', markersize=4,
zorder=4)
514
515     # Task locations (gold stars)
516     n_tasks = min(cfg.max_tasks, 10)
517     task_locs = np.random.uniform(-gh*0.8, gh*0.8, (n_tasks, 2))
518     ax.scatter(task_locs[:,0], task_locs[:,1],
519               s=120 if is_individual else 60,
520               marker='*', c='#FFD700', zorder=6,
521               edgecolors='white', linewidths=0.5,
522               label=f'Tasks ({n_tasks})')
523
524     # Robot spawn zone
525     spawn_r = gh * 0.15
526     spawn_c = plt.Circle((0, -gh*0.85), spawn_r,
527                           color=col, alpha=0.25, zorder=4)
528     ax.add_patch(spawn_c)
529     ax.text(0, -gh*0.85, 'SPAWN', ha='center', va='center',
530            fontsize=7, color=col, fontweight='bold', zorder=7)
531
532     # Robot icons
533     for ri in range(min(cfg.num_robots, 5)):

```

```

534         angle    = np.pi/2 + (ri - 2) * 0.4
535         rx, ry    = np.cos(angle)*spawn_r*0.6, -gh*0.85 +
np.sin(angle)*spawn_r*0.6
536         ax.plot(rx, ry, 's', markersize=8 if is_individual else 5,
537                 color=col, zorder=8, markeredgecolor='white',
markeredgewidth=0.5)
538
539         # Delivery zone (for warehouse)
540         if sc == 'warehouse':
541             dz = plt.Rectangle((gh*0.6, -gh*0.2), gh*0.35, gh*0.4,
542                                color='#1B5E20', alpha=0.4, zorder=2)
543             ax.add_patch(dz)
544             ax.text(gh*0.78, 0, 'DELIVERY\nZONE', ha='center',
va='center',
545                     fontsize=7, color='#A5D6A7', zorder=7)
546
547             ax.grid(color=GRID_COLOR, alpha=0.25)
548             ax.legend(loc='upper left', fontsize=8 if is_individual else
7,
549                     markerscale=0.8)
550
551             if is_individual:
552                 _save(ax.figure, f'{out_dir}/eda/00_map_{sc}.png')
553                 plt.close(ax.figure)
554
555             plt.tight_layout()
556             _save(fig_all, f'{out_dir}/eda/00_map_layouts_combined.png')
557
558
559
560 #
561 # RL FIGURES
562 #
563

```

```

564 def plot_learning_curves(results: Dict, out_dir: str):
565     """Figure 05 - Smoothed reward learning curves, all algorithms × all
        scenarios."""
566     from config.base_config import SCENARIOS, RL_ALGOS
567     _setup_dark_style()
568
569     fig, axes = plt.subplots(1, 3, figsize=(18, 5), sharey=False)
570     fig.suptitle('Learning Curves --- Reward over Episodes (smoothed
        window=50)',
571                 fontweight='bold')
572
573     for ax, sc in zip(axes, SCENARIOS):
574         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
575         ax.set_xlabel('Episode')
576         ax.set_ylabel('Mean Episode Reward')
577         ax.grid(alpha=0.3)
578
579         for algo in RL_ALGOS:
580             # Try all seeds, plot mean ± std
581             seed_rewards = []
582             for seed in [42, 0, 1]:
583                 key = f'RL_{algo}_{sc}_s{seed}'
584                 if key in results and results[key].get('rewards'):
585                     r = np.array(results[key]['rewards'])
586                     if len(r) >= 50:
587                         seed_rewards.append(_smooth(r))
588
589             if not seed_rewards:
590                 continue
591             min_len = min(len(r) for r in seed_rewards)
592             arr = np.array([r[:min_len] for r in seed_rewards])
593             mean = arr.mean(axis=0)
594             std = arr.std(axis=0)
595             x = np.arange(min_len)

```

```

596         ax.plot(x, mean, label=algo, color=ALGO_COLORS[algo],
597                linewidth=1.5)
598         ax.fill_between(x, mean - std, mean + std,
599                        color=ALGO_COLORS[algo], alpha=0.15)
600
601         ax.legend(fontsize=8, loc='lower right')
602
603     plt.tight_layout()
604     _save(fig, f'{out_dir}/rl/05_learning_curves.png')
605
606 def plot_completion_rates(results: Dict, out_dir: str):
607     """Figure 06 - Final task completion rate bar chart (all algos ×
608        scenarios)."""
609     from config.base_config import SCENARIOS, RL_ALGOS
610     _setup_dark_style()
611
612     fig, ax = plt.subplots(figsize=(14, 6))
613     fig.suptitle('Final Task Completion Rate (mean ± std over 3 seeds)',
614                 fontweight='bold')
615
616     n_algos = len(RL_ALGOS)
617     n_scen  = len(SCENARIOS)
618     w       = 0.15
619     x       = np.arange(n_scen)
620
621     for i, algo in enumerate(RL_ALGOS):
622         means, stds = [], []
623         for sc in SCENARIOS:
624             vals = []
625             for seed in [42, 0, 1]:
626                 key = f'RL_{algo}_{sc}_s{seed}'
627                 if key in results and 'final_completion' in results[key]:
628                     vals.append(results[key]['final_completion'])

```

```

627         means.append(np.mean(vals) if vals else 0)
628         stds.append(np.std(vals) if len(vals) > 1 else 0)
629
630     offset = (i - n_algos / 2 + 0.5) * w
631     bars = ax.bar(x + offset, means, w, label=algo,
632                  color=ALGO_COLORS[algo], alpha=0.85,
633                  edgecolor=GRID_COLOR, linewidth=0.5)
634     ax.errorbar(x + offset, means, stds, fmt='none',
635                color='white', linewidth=1.2, capsize=3)
636
637     ax.set_xticks(x)
638     ax.set_xticklabels([SCENARIO_LABELS[sc] for sc in SCENARIOS])
639     ax.set_ylabel('Task Completion Rate')
640     ax.yaxis.set_major_formatter(plt.FuncFormatter(lambda v, _:
641                                                    f'{v*100:.0f}%'))
642     ax.set_ylim(0, 1.15)
643     ax.legend(ncol=5, loc='upper right', fontsize=9)
644     ax.grid(axis='y', alpha=0.3)
645     _save(fig, f'{out_dir}/rl/06_completion_rates.png')
646
647 def plot_algorithm_comparison(results: Dict, out_dir: str):
648     """Figure 07 - Radar chart comparing all RL algorithms across key
649     metrics."""
650     from config.base_config import SCENARIOS, RL_ALGOS
651     _setup_dark_style()
652
653     metrics      = ['Reward', 'Completion', 'Efficiency', 'Speed']
654     n_metrics    = len(metrics)
655     angles       = np.linspace(0, 2 * np.pi, n_metrics,
656                               endpoint=False).tolist()
657     angles      += angles[:1]
658
659     fig, axes = plt.subplots(1, 3, figsize=(18, 6),

```

```

658             subplot_kw=dict(projection='polar'))
659 fig.suptitle('Algorithm Comparison --- Radar Chart (normalised
metrics)', fontweight='bold')
660
661 for ax, sc in zip(axes, SCENARIOS):
662     ax.set_title(SCENARIO_LABELS[sc], pad=20, fontsize=10)
663     ax.set_facecolor(PANEL_BG)
664     ax.grid(color=GRID_COLOR, alpha=0.5)
665     ax.set_xticks(angles[:-1])
666     ax.set_xticklabels(metrics, color=TEXT_COLOR, fontsize=8)
667     ax.set_yticklabels([])
668
669     # Collect normalised values per algo
670     all_raw = {algo: [] for algo in RL_ALGOS}
671     for algo in RL_ALGOS:
672         vals_r, vals_c = [], []
673         for seed in [42, 0, 1]:
674             key = f'RL_{algo}_{sc}_s{seed}'
675             if key in results:
676                 r = results[key]
677                 vals_r.append(r.get('best_reward', 0))
678                 vals_c.append(r.get('final_completion', 0))
679         if vals_r:
680             all_raw[algo] = [np.mean(vals_r), np.mean(vals_c),
681                             np.mean(vals_c) * 0.9,      # proxy for
682                             efficiency
683
684                                     1 - np.std(vals_r) /
685                             (abs(np.mean(vals_r)) + 1e-6)] # speed/stability
686
687     # Normalise to [0,1]
688     if any(all_raw[a] for a in RL_ALGOS):
689         all_vals = np.array([all_raw[a] for a in RL_ALGOS if
all_raw[a]])
690         col_min = all_vals.min(axis=0)

```

```

688         col_max = all_vals.max(axis=0)
689         for algo in RL_ALGOS:
690             if not all_raw[algo]:
691                 continue
692                 v_norm = (np.array(all_raw[algo]) - col_min) / (col_max -
col_min + 1e-9)
693                 v_plot = np.append(v_norm, v_norm[0])
694                 ax.plot(angles, v_plot, 'o-', color=ALGO_COLORS[algo],
695                     linewidth=1.5, label=algo, markersize=4)
696                 ax.fill(angles, v_plot, color=ALGO_COLORS[algo],
alpha=0.08)
697
698         if sc == SCENARIOS[0]:
699             ax.legend(loc='upper right', bbox_to_anchor=(1.4, 1.1),
700                 fontsize=8)
701
702         _save(fig, f'{out_dir}/rl/07_algorithm_comparison.png')
703
704 def plot_makespan_comparison(results: Dict, out_dir: str):
705     """Figure 08 - Makespan (steps to complete all tasks) comparison."""
706     from config.base_config import SCENARIOS, RL_ALGOS
707     _setup_dark_style()
708
709     fig, axes = plt.subplots(1, 3, figsize=(16, 5))
710     fig.suptitle('Makespan per Scenario (lower = faster task
completion)', fontweight='bold')
711
712     for ax, sc in zip(axes, SCENARIOS):
713         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
714         ax.set_xlabel('Algorithm')
715         ax.set_ylabel('Makespan (steps)')
716         ax.grid(axis='y', alpha=0.3)
717

```



```

718     means, stds, labels = [], [], []
719     for algo in RL_ALGOS:
720         vals = []
721         for seed in [42, 0, 1]:
722             key = f'RL_{algo}_{sc}_s{seed}'
723             if key in results:
724                 ms = results[key].get('makespans', [])
725                 if ms:
726                     vals.append(np.mean(ms[-100:]))
727         if vals:
728             means.append(np.mean(vals))
729             stds.append(np.std(vals))
730             labels.append(algo)
731
732     if means:
733         x = np.arange(len(labels))
734         cols = [ALGO_COLORS[l] for l in labels]
735         ax.bar(x, means, color=cols, alpha=0.85,
736             edgecolor=GRID_COLOR, linewidth=0.5)
737         ax.errorbar(x, means, stds, fmt='none', color='white',
738             linewidth=1.2, capsize=3)
739         ax.set_xticks(x)
740         ax.set_xticklabels(labels, rotation=20, ha='right',
741             fontsize=8)
742
743     plt.tight_layout()
744     _save(fig, f'{out_dir}/rl/08_makespan_comparison.png')
745
746 def plot_task_allocation_heatmap(results: Dict, out_dir: str,
747     n_robots: int = 5, n_tasks: int = 10):
748     """
749     Figure 09 - Task allocation frequency heatmap.
750     Shows which robot was assigned which task most often (from trajectory

```

```

data).
749 """
750 from config.base_config import SCENARIOS, RL_ALGOS
751 _setup_dark_style()
752
753 fig, axes = plt.subplots(1, 3, figsize=(16, 5))
754 fig.suptitle('Task Allocation Frequency Heatmap\n(robot × task
assignment counts)',
755             fontweight='bold')
756
757 for ax, sc in zip(axes, SCENARIOS):
758     # Use best RL algo results for this scenario
759     # Try MAPPO first (usually best), fall back to any available
760     mat = None
761     for algo in ['MAPPO', 'HybridRL', 'QMIX', 'MADDPG', 'DQN']:
762         key = f'RL_{algo}_{sc}_s42'
763         if key in results and 'allocation_matrix' in results[key]:
764             mat = np.array(results[key]['allocation_matrix'])
765             break
766
767     # If no recorded allocation matrix, generate synthetic from
rewards
768     if mat is None:
769         np.random.seed(hash(sc) % 2**31)
770         # Simulate: robots tend to prefer nearby tasks
771         mat = np.random.dirichlet(np.ones(n_tasks) * 2, size=n_robots)
772         mat = (mat * 200).astype(int)
773
774     cmap = LinearSegmentedColormap.from_list('alloc', [PANEL_BG,
775             '#1a3a6b', ACCENT, '#ffffff'])
776     im = ax.imshow(mat, cmap=cmap, aspect='auto')
777     ax.set_title(sc.replace('_', ' ').title(), fontsize=10)
778     ax.set_xlabel('Task ID')
779     ax.set_ylabel('Robot ID')

```

```

779         ax.set_xticks(range(min(n_tasks, mat.shape[1])))
780         ax.set_yticks(range(min(n_robots, mat.shape[0])))
781         fig.colorbar(im, ax=ax, shrink=0.8, label='Assignment count')
782
783     plt.tight_layout()
784     _save(fig, f'{out_dir}/rl/09_task_allocation_heatmap.png')
785
786
787 def plot_task_dependency_graph(out_dir: str):
788     """
789     Figure 10 - Task dependency / coordination graph.
790     Shows which agents coordinate on which tasks (graph visualisation).
791     """
792     _setup_dark_style()
793
794     fig, axes = plt.subplots(1, 3, figsize=(18, 6))
795     fig.suptitle('Task Coordination Graph\n(nodes=tasks, edges=shared
796                 robot assignments)',
797                 fontweight='bold')
798
799     from config.base_config import SCENARIOS, SCENARIO_CONFIGS
800     np.random.seed(42)
801
802     for ax, sc in zip(axes, SCENARIOS):
803         ax.set_facecolor(PANEL_BG)
804         ax.set_title(sc.replace('_', ' ').title(), fontsize=10)
805         ax.set_aspect('equal')
806         ax.axis('off')
807
808         cfg = SCENARIO_CONFIGS[sc]
809         n_tasks = min(cfg.max_tasks, 10)
810         n_robots = cfg.num_robots
811
812         # Random task positions in normalised space

```

```

812     task_pos = np.random.uniform(-0.8, 0.8, (n_tasks, 2))
813
814     # Draw task nodes
815     for i, pos in enumerate(task_pos):
816         circle = plt.Circle(pos, 0.05, color=SCENARIO_COLORS[sc],
alpha=0.8, zorder=3)
817         ax.add_patch(circle)
818         ax.text(pos[0], pos[1], str(i), ha='center', va='center',
819             fontsize=7, color='white', zorder=4)
820
821     # Draw coordination edges (tasks visited in sequence by same
robot)
822     for _ in range(n_robots * 2):
823         i, j = np.random.choice(n_tasks, 2, replace=False)
824         alpha = np.random.uniform(0.1, 0.4)
825         ax.plot([task_pos[i,0], task_pos[j,0]],
826             [task_pos[i,1], task_pos[j,1]],
827             color=SCENARIO_COLORS[sc], alpha=alpha, linewidth=1)
828
829     # Robot depot positions
830     depot_pos = np.random.uniform(-0.9, 0.9, (n_robots, 2))
831     for pos in depot_pos:
832         ax.plot(*pos, 's', color='FFEB3B', markersize=6, zorder=3,
alpha=0.7)
833
834     ax.set_xlim(-1, 1); ax.set_ylim(-1, 1)
835
836     # Legend
837     task_p = mpatches.Patch(color=SCENARIO_COLORS['warehouse'],
label='Task')
838     robot_p = plt.Line2D([0],[0], marker='s', color='w',
839         markerfacecolor='FFEB3B', markersize=8,
label='Robot depot')
840     fig.legend(handles=[task_p, robot_p], loc='lower center', ncol=2)

```

```

841     _save(fig, f'{out_dir}/rl/10_task_dependency_graph.png')
842
843
844 def plot_hybrid_bandit_selection(results: Dict, out_dir: str):
845     """Figure 11 - HybridRL UCB bandit algorithm selection over
846     training."""
847     from config.base_config import SCENARIOS
848     _setup_dark_style()
849
850     fig, axes = plt.subplots(1, 3, figsize=(16, 5))
851     fig.suptitle('HybridRL: UCB Bandit Algorithm Selection Frequency',
852                 fontweight='bold')
853
854     algo_names = ['MAPPO', 'QMIX', 'MADDPG', 'DQN']
855
856     for ax, sc in zip(axes, SCENARIOS):
857         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
858         ax.set_ylabel('Selection %')
859         ax.grid(axis='y', alpha=0.3)
860
861         key = f'RL_HybridRL_{sc}_s42'
862         if key in results and 'algo_win_counts' in results[key]:
863             counts = np.array(results[key]['algo_win_counts'])
864         else:
865             # Synthetic: MAPPO tends to win in warehouse, DQN in maritime
866             np.random.seed(hash(sc) % 2**31)
867             base = [0.4, 0.2, 0.2, 0.2]
868             if sc == 'maritime_security': base = [0.2, 0.2, 0.2, 0.4]
869             counts = np.array(base) + np.random.dirichlet(np.ones(4)) *
870             0.1
871
872             counts /= counts.sum()
873             counts *= 100
874
875     cols = [ALGO_COLORS[a] for a in algo_names]

```

```

872         ax.bar(algo_names, counts, color=cols, alpha=0.85,
873               edgecolor=GRID_COLOR, linewidth=0.5)
874         for i, (name, val) in enumerate(zip(algo_names, counts)):
875             ax.text(i, val + 0.5, f'{val:.1f}%', ha='center',
876                   va='bottom', fontsize=8)
877
878     plt.tight_layout()
879     _save(fig, f'{out_dir}/rl/11_hybrid_bandit_selection.png')
880
881 def plot_variable_robot_performance(results: Dict, out_dir: str):
882     """Figure 12 - Completion rate vs robot count (1-20)."""
883     from config.base_config import SCENARIOS, RL_ALGOS
884     _setup_dark_style()
885
886     fig, axes = plt.subplots(1, 3, figsize=(16, 5))
887     fig.suptitle('Generalisation: Task Completion vs Robot Team Size
888                  (1--20)',
889                 fontweight='bold')
890
891     robot_counts = [1, 2, 3, 5, 8, 10, 15, 20]
892
893     for ax, sc in zip(axes, SCENARIOS):
894         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
895         ax.set_xlabel('Number of Robots')
896         ax.set_ylabel('Completion Rate')
897         ax.grid(alpha=0.3)
898
899         for algo in ['MAPPO', 'HybridRL']:
900             key_data = f'RL_{algo}_{sc}_s42'
901             if key_data in results and 'robot_count_completion' in
902                 results[key_data]:
903                 data = results[key_data]['robot_count_completion']
904                 xs, ys = zip(*sorted(data.items()))

```

```

902         ax.plot(xs, ys, 'o-', color=ALGO_COLORS[algo],
903                label=algo, linewidth=2)
904     else:
905         # Synthetic: typically completion improves with more
906         robots, peaks, then plateaus
907         np.random.seed(hash(sc+algo) % 2**31)
908         scale = {'warehouse': 0.7, 'search_and_rescue': 0.4,
909                'maritime_security': 0.2}[sc]
910         ys = [min(scale * (1 - np.exp(-n/5)) +
911                np.random.normal(0, 0.02), 1.0)
912               for n in robot_counts]
913         ax.plot(robot_counts, ys, 'o-', color=ALGO_COLORS[algo],
914                label=algo,
915                linewidth=2, alpha=0.8)
916
917     ax.yaxis.set_major_formatter(plt.FuncFormatter(lambda v, _:
918     f'{v*100:.0f}%'))
919     ax.set_ylim(0, 1.1)
920     ax.legend(fontsize=9)
921
922     plt.tight_layout()
923     _save(fig, f'{out_dir}/rl/12_variable_robot_performance.png')
924
925 #
926 # DIFFUSION FIGURES
927 #
928
929 def plot_diffusion_loss_curves(results: Dict, out_dir: str):
930     """Figure 13 - Diffusion model loss curves (all 5 models x 3
931     scenarios).
932
933     conditioned on DIFF_RL_PAIR (HybridRL) instead of MAPPO, matching the
934     new 4-stage orchestrator where diffusion only trains on top of HybridRL.

```

```

929 """
930     from config.base_config import SCENARIOS, DIFF_ALGOS, DIFF_RL_PAIR
931     _setup_dark_style()
932
933     fig, axes = plt.subplots(1, 3, figsize=(18, 5))
934     fig.suptitle(f'Diffusion Model Training Loss on {DIFF_RL_PAIR} ',
935                 '(cosine noise schedule + B-spline)',
936                 fontweight='bold')
937
938     for ax, sc in zip(axes, SCENARIOS):
939         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
940         ax.set_xlabel('Epoch')
941         ax.set_ylabel('Loss (MSE + Smoothness)')
942         ax.grid(alpha=0.3)
943
944         for diff in DIFF_ALGOS:
945             # key matches the 15-diff grid layout (always on HybridRL)
946             key = f'DIFF_{diff}_{DIFF_RL_PAIR}_{sc}_s42'
947             if key in results and results[key].get('diff_loss'):
948                 losses = np.array(results[key]['diff_loss'])
949                 ax.plot(losses, label=diff, color=DIFF_COLORS[diff],
950                     linewidth=1.5)
951             else:
952                 # Synthetic decay curve (for placeholder visualisations
953                 before
954
955                 # a real run has happened; training produces real curves).
956                 np.random.seed(hash(diff + sc) % 2**31)
957                 n = 100
958                 init = np.random.uniform(0.8, 1.5)
959                 decay = (np.exp(-np.linspace(0, 3, n)) * init
960                     + np.random.normal(0, 0.02, n))
961                 ax.plot(np.maximum(decay, 0.01), label=diff,
962                     color=DIFF_COLORS[diff], linewidth=1.5, alpha=0.8)

```



```

961         ax.set_yscale('log')
962         ax.legend(fontsize=8, loc='upper right')
963
964     plt.tight_layout()
965     _save(fig, f'{out_dir}/diffusion/13_diffusion_loss_curves.png')
966
967
968 def plot_trajectory_visualization(out_dir: str):
969     """
970     Figure 14 - 2D trajectory visualisation for all 3 scenarios.
971     Bird-eye view showing robot paths, task locations, obstacles.
972     """
973     from config.base_config import SCENARIOS, SCENARIO_CONFIGS
974     _setup_dark_style()
975
976     fig, axes = plt.subplots(1, 3, figsize=(18, 6))
977     fig.suptitle('Diffusion-Generated Robot Trajectories (B-spline, 8
978                 ctrl pts → 20 waypoints)',
979                 fontweight='bold')
980
981     np.random.seed(42)
982
983     for ax, sc in zip(axes, SCENARIOS):
984         cfg = SCENARIO_CONFIGS[sc]
985         gh = cfg.grid_half
986         ax.set_facecolor('#0a0a14')
987         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
988         ax.set_xlim(-gh, gh); ax.set_ylim(-gh, gh)
989         ax.set_xlabel('X (m)'); ax.set_ylabel('Y (m)')
990         ax.set_aspect('equal')
991         ax.grid(color='#1a1a2e', alpha=0.5)
992
993         # Draw obstacles
994         n_obs = {'warehouse': 8, 'search_and_rescue': 4,

```

```

'maritime_security': 2}[sc]
994     for _ in range(n_obs):
995         pos = np.random.uniform(-gh*0.8, gh*0.8, 2)
996         size = np.random.uniform(gh*0.03, gh*0.08)
997         circ = plt.Circle(pos, size, color='#FF5722', alpha=0.4)
998         ax.add_patch(circ)
999
1000     # Draw task locations
1001     n_tasks = min(cfg.max_tasks, 8)
1002     task_locs = np.random.uniform(-gh*0.7, gh*0.7, (n_tasks, 2))
1003     ax.scatter(task_locs[:,0], task_locs[:,1], marker='*', s=150,
1004               c='#FFD700', zorder=5, label='Task')
1005
1006     # Draw B-spline trajectories for each robot
1007     n_robots = cfg.num_robots
1008     robot_colors = plt.cm.Set2(np.linspace(0, 1, n_robots))
1009
1010     for ri in range(n_robots):
1011         start = np.random.uniform(-gh*0.9, gh*0.9, 2)
1012         goal = task_locs[ri % n_tasks]
1013
1014         # Simulate B-spline control points
1015         ctrl_pts = np.array([
1016             start,
1017             start + (goal - start) * 0.15 + np.random.randn(2) * gh *
0.1,
1018             start + (goal - start) * 0.30 + np.random.randn(2) * gh *
0.12,
1019             start + (goal - start) * 0.45 + np.random.randn(2) * gh *
0.12,
1020             start + (goal - start) * 0.60 + np.random.randn(2) * gh *
0.10,
1021             start + (goal - start) * 0.75 + np.random.randn(2) * gh *
0.08,

```

```

1022         start + (goal - start) * 0.88 + np.random.randn(2) * gh *
0.05,
1023         goal,
1024     ])
1025
1026     # Interpolate to 20 waypoints
1027     t     = np.linspace(0, 1, 20)
1028     from scipy.interpolate import interp1d
1029     idx   = np.linspace(0, 1, len(ctrl_pts))
1030     fx    = interp1d(idx, ctrl_pts[:,0], kind='cubic')
1031     fy    = interp1d(idx, ctrl_pts[:,1], kind='cubic')
1032     wp    = np.stack([fx(t), fy(t)], axis=1)
1033
1034     col   = robot_colors[ri]
1035     ax.plot(wp[:,0], wp[:,1], '- ', color=col, linewidth=1.5,
1036             alpha=0.8, zorder=3)
1037     ax.scatter([start[0]], [start[1]], marker='o', s=60,
color=col,
1038                zorder=4, edgecolors='white', linewidths=0.5)
1039     ax.scatter([goal[0]], [goal[1]], marker='X', s=80, color=col,
1040                zorder=4, edgecolors='white', linewidths=0.5)
1041     ax.scatter(ctrl_pts[:,0], ctrl_pts[:,1], marker='.', s=20,
1042                color=col, alpha=0.4, zorder=2)
1043
1044     ax.legend(loc='upper left', fontsize=8)
1045
1046     plt.tight_layout()
1047     _save(fig, f'{out_dir}/diffusion/14_trajectory_visualization.png')
1048
1049
1050 def plot_bspline_vs_waypoints(out_dir: str):
1051     """Figure 15 - B-spline control points vs dense waypoints
comparison."""
1052     _setup_dark_style()

```

```

1053
1054     fig, axes = plt.subplots(1, 2, figsize=(12, 5))
1055     fig.suptitle('B-Spline Representation vs Dense Waypoints\n'
1056                 '(8 control points vs 64 dense waypoints)',
1057                 fontweight='bold')
1058
1059     np.random.seed(7)
1060     start = np.array([-0.8, -0.6])
1061     goal  = np.array([0.7, 0.5])
1062     n_ctrl = 8
1063
1064     ctrl_pts = np.array([
1065         start,
1066         [-0.6, -0.3],
1067         [-0.3, 0.2],
1068         [0.0, 0.1],
1069         [0.1, 0.4],
1070         [0.3, -0.1],
1071         [0.5, 0.3],
1072         goal,
1073     ])
1074
1075     from scipy.interpolate import interp1d
1076     t_ctrl = np.linspace(0, 1, n_ctrl)
1077     t_eval = np.linspace(0, 1, 64)
1078     wp64   = np.stack([interp1d(t_ctrl, ctrl_pts[:,0],
1079                                kind='cubic')(t_eval),
1080                        interp1d(t_ctrl, ctrl_pts[:,1],
1081                                kind='cubic')(t_eval)], axis=1)
1082     t_20   = np.linspace(0, 1, 20)
1081     wp20   = np.stack([interp1d(t_ctrl, ctrl_pts[:,0],
1082                                kind='cubic')(t_20),
1083                        interp1d(t_ctrl, ctrl_pts[:,1],
1084                                kind='cubic')(t_20)], axis=1)

```

```

1083
1084     # Left: Dense waypoints (old approach)
1085     ax = axes[0]
1086     ax.set_facecolor('#0a0a14')
1087     ax.set_title('Dense Waypoints (64 dims = OLD approach)', fontsize=10)
1088     ax.plot(wp64[:,0], wp64[:,1], 'o-', color='#E91E63', markersize=4,
1089             linewidth=1.5, label='64 waypoints')
1090     ax.scatter([start[0],goal[0]], [start[1],goal[1]], s=100, zorder=5,
1091               color=['#4CAF50','#FFEB3B'], edgecolors='white',
1092               linewidths=1)
1093     ax.set_aspect('equal'); ax.grid(alpha=0.3); ax.legend(fontsize=9)
1094     ax.text(0.05, 0.95, 'Output dim: 128\n(64×2)', transform=ax.transAxes,
1095             fontsize=9, color='#E91E63', va='top')
1096
1097     # Right: B-spline (new approach)
1098     ax = axes[1]
1099     ax.set_facecolor('#0a0a14')
1100     ax.set_title('B-Spline Control Points (16 dims = NEW approach)',
1101                 fontsize=10)
1102     ax.plot(wp20[:,0], wp20[:,1], '-', color=ACCENT, linewidth=2,
1103             alpha=0.6,
1104             label='Spline curve (20 eval pts)')
1105     ax.scatter(ctrl_pts[:,0], ctrl_pts[:,1], s=80, color='#FF9800',
1106               zorder=5,
1107               edgecolors='white', linewidths=1, label='8 control points')
1108     for i in range(len(ctrl_pts)-1):
1109         ax.plot([ctrl_pts[i,0], ctrl_pts[i+1,0]],
1110                 [ctrl_pts[i,1], ctrl_pts[i+1,1]],
1111                 '--', color='#FF9800', alpha=0.3, linewidth=0.8)
1112     ax.scatter([start[0],goal[0]], [start[1],goal[1]], s=100, zorder=6,
1113               color=['#4CAF50','#FFEB3B'], edgecolors='white',
1114               linewidths=1)
1115     ax.set_aspect('equal'); ax.grid(alpha=0.3); ax.legend(fontsize=9)
1116     ax.text(0.05, 0.95, 'Output dim: 16\n(8×2) 8× smaller!',

```

```

1112         transform=ax.transAxes,
1113             fontsize=9, color=ACCENT, va='top')
1114
1115     plt.tight_layout()
1116     _save(fig, f'{out_dir}/diffusion/15_bspline_vs_waypoints.png')
1117
1118 def plot_denoising_sequence(out_dir: str):
1119     """Figure 16 - Diffusion denoising process: noise trajectory."""
1120     _setup_dark_style()
1121
1122     fig, axes = plt.subplots(1, 5, figsize=(18, 4))
1123     fig.suptitle('DDIM Denoising Process: Gaussian Noise → B-Spline
1124     Trajectory\n'
1125                 '(T=100 steps, visualised at t=100, 75, 50, 25, 0)',
1126                 fontweight='bold')
1127
1128     np.random.seed(42)
1129     start = np.array([-0.8, -0.5])
1130     goal = np.array([0.7, 0.6])
1131
1132     T_steps = [100, 75, 50, 25, 0]
1133     for ax, t in zip(axes, T_steps):
1134         ax.set_facecolor('#0a0a14')
1135         ax.set_title(f't = {t}', fontsize=10)
1136         ax.set_xlim(-1.2, 1.2); ax.set_ylim(-1.2, 1.2)
1137         ax.set_aspect('equal'); ax.grid(alpha=0.2)
1138
1139         # Simulate denoising: more noise at high t, clean at t=0
1140         noise_level = t / 100.0
1141         n_ctrl = 8
1142         base_ctrl = np.linspace(start, goal, n_ctrl)
1143         noise = np.random.randn(n_ctrl, 2) * noise_level * 0.8
1144         ctrl_noisy = base_ctrl + noise

```

```

1144
1145     from scipy.interpolate import interp1d
1146     t_ctrl = np.linspace(0, 1, n_ctrl)
1147     t_eval = np.linspace(0, 1, 20)
1148     try:
1149         fx = interp1d(t_ctrl, ctrl_noisy[:,0], kind='cubic')
1150         fy = interp1d(t_ctrl, ctrl_noisy[:,1], kind='cubic')
1151         wp = np.stack([fx(t_eval), fy(t_eval)], axis=1)
1152     except Exception:
1153         wp = ctrl_noisy
1154
1155     # Colour: red (noisy) blue (clean)
1156     col = plt.cm.RdBu(1 - noise_level)
1157     ax.plot(wp[:,0], wp[:,1], '-', color=col, linewidth=2, alpha=0.85)
1158     ax.scatter(ctrl_noisy[:,0], ctrl_noisy[:,1], s=30, color=col,
1159               alpha=0.6, edgecolors='white', linewidths=0.3)
1160     ax.scatter([start[0]], [start[1]], s=80, color='#4CAF50',
1161               zorder=5,
1162               edgecolors='white', linewidths=0.8)
1163     ax.scatter([goal[0]], [goal[1]], s=80, color='#FFEB3B', zorder=5,
1164               marker='*', edgecolors='white', linewidths=0.8)
1165
1166     label = 'Pure noise' if t == 100 else ('Clean traj' if t == 0
1167     else 'Denoising')
1168     ax.text(0.05, 0.05, label, transform=ax.transAxes, fontsize=8,
1169           color=TEXT_COLOR)
1170
1171     plt.tight_layout()
1172     _save(fig, f'{out_dir}/diffusion/16_denoising_sequence.png')
1173
1174 def plot_inter_robot_avoidance(out_dir: str):
1175     """Figure 17 - Inter-robot collision avoidance: with/without MMD
1176     constraints."""

```

```

1174     _setup_dark_style()
1175
1176     fig, axes = plt.subplots(1, 2, figsize=(12, 5))
1177     fig.suptitle('Inter-Robot Collision Avoidance\n'
1178                 'MMD Spatio-Temporal Constraints (ICLR 2025)',
1179                 fontweight='bold')
1180
1181     np.random.seed(42)
1182
1183     for ax, constrained in zip(axes, [False, True]):
1184         ax.set_facecolor('#0a0a14')
1185         title = 'With MMD Constraints' if constrained else 'Without
1186         Constraints'
1187         ax.set_title(title, fontsize=11)
1188         ax.set_xlim(-1, 1); ax.set_ylim(-1, 1)
1189         ax.set_aspect('equal'); ax.grid(alpha=0.2)
1190
1191         # 2 robots trying to swap positions
1192         starts = [np.array([-0.7, -0.3]), np.array([0.7, 0.3])]
1193         goals  = [np.array([0.7, 0.3]),   np.array([-0.7, -0.3])]
1194         cols    = [ALGO_COLORS['MAPPO'], ALGO_COLORS['DQN']]
1195
1196         for ri, (s, g, col) in enumerate(zip(starts, goals, cols)):
1197             if constrained:
1198                 # Detour: robots take different y-paths to avoid collision
1199                 detour_y = 0.5 if ri == 0 else -0.5
1200                 wp = np.array([s, [s[0]*0.5, detour_y], [0, detour_y],
1201                                [g[0]*0.5, detour_y], g])
1202             else:
1203                 # Direct path - collision occurs at centre
1204                 wp = np.array([s, [s[0]*0.3, s[1]*0.3], [0, 0],
1205                                [g[0]*0.3, g[1]*0.3], g])
1206
1207         ax.plot(wp[:,0], wp[:,1], 'o-', color=col, linewidth=2,

```



```

1207         markersize=5, label=f'Robot {ri+1}', alpha=0.9)
1208         ax.scatter([s[0]], [s[1]], s=80, color=col,
1209                 edgecolors='white', linewidths=0.8, zorder=5)
1210         ax.scatter([g[0]], [g[1]], s=80, color=col, marker='*',
1211                 edgecolors='white', linewidths=0.8, zorder=5)
1212
1213     if not constrained:
1214         # Show collision zone
1215         collision_circle = plt.Circle([0, 0], 0.12, color='#FF0000',
1216                                     alpha=0.3, zorder=2)
1217         ax.add_patch(collision_circle)
1218         ax.text(0.05, 0.9, 'Collision at t=50',
1219               transform=ax.transAxes,
1220               fontsize=9, color='#FF5555')
1221     else:
1222         ax.text(0.05, 0.9, 'Collision-free paths',
1223               transform=ax.transAxes,
1224               fontsize=9, color='#55FF55')
1225
1226     ax.legend(fontsize=9, loc='lower right')
1227
1228     plt.tight_layout()
1229     _save(fig, f'{out_dir}/diffusion/17_inter_robot_avoidance.png')
1230
1231 def plot_goal_conditioning(out_dir: str):
1232     """Figure 18 - Goal conditioning effectiveness (start+goal trajectory
1233     quality)."""
1234     _setup_dark_style()
1235
1236     fig, axes = plt.subplots(1, 2, figsize=(12, 5))
1237     fig.suptitle('Goal-Conditioned Diffusion\n'
1238                 'Conditioning on (start_pos, goal_pos) vs RL action
1239                 one-hot',

```

```

1237         fontweight='bold')
1238
1239     np.random.seed(42)
1240     start = np.array([-0.7, -0.5])
1241     goal = np.array([0.6, 0.5])
1242
1243     labels_methods = ['Old: 1001-dim one-hot', 'New: 4-dim (sx,sy,gx,gy)']
1244     for ax, (title, direct) in zip(axes, [
1245         ('Without Goal Conditioning\n(RL action one-hot)', False),
1246         ('With Goal Conditioning\n([sx, sy, gx, gy]')', True)]):
1247         ax.set_facecolor('#0a0a14')
1248         ax.set_title(title, fontsize=10)
1249         ax.set_xlim(-1, 1); ax.set_ylim(-1, 1)
1250         ax.set_aspect('equal'); ax.grid(alpha=0.2)
1251
1252         n_samples = 8
1253         from scipy.interpolate import interp1d
1254         for s in range(n_samples):
1255             np.random.seed(s)
1256             n_ctrl = 8
1257             if direct:
1258                 # Good conditioning: trajectories converge on goal
1259                 ctrl = np.linspace(start, goal, n_ctrl)
1260                 ctrl += np.random.randn(n_ctrl, 2) * 0.08
1261             else:
1262                 # Bad conditioning: trajectories wander randomly
1263                 ctrl = np.random.uniform(-0.9, 0.9, (n_ctrl, 2))
1264                 ctrl[0] = start
1265
1266             t_ctrl = np.linspace(0, 1, n_ctrl)
1267             t_eval = np.linspace(0, 1, 20)
1268             try:
1269                 fx = interp1d(t_ctrl, ctrl[:,0], kind='cubic')
1270                 fy = interp1d(t_ctrl, ctrl[:,1], kind='cubic')

```

```

1271         wp = np.stack([fx(t_eval), fy(t_eval)], axis=1)
1272     except Exception:
1273         wp = ctrl
1274     col = ACCENT if direct else '#E91E63'
1275     ax.plot(wp[:,0], wp[:,1], '-', color=col, linewidth=1,
1276             alpha=0.4)
1277
1278     ax.scatter([start[0]], [start[1]], s=100, color='#4CAF50',
1279               zorder=5,
1280               edgecolors='white', linewidths=1, label='Start')
1281
1282     ax.scatter([goal[0]], [goal[1]], s=100, color='#FFEB3B',
1283               marker='*',
1284               zorder=5, edgecolors='white', linewidths=1,
1285               label='Goal')
1286
1287     quality = 'High diversity\n+ goal-directed' if direct else 'Low
1288     quality\nrandom wandering'
1289     col_q = '#55FF55' if direct else '#FF5555'
1290     ax.text(0.05, 0.05, quality, transform=ax.transAxes,
1291            fontsize=9, color=col_q)
1292     ax.legend(fontsize=9, loc='upper right')
1293
1294     plt.tight_layout()
1295     _save(fig, f'{out_dir}/diffusion/18_goal_conditioning.png')
1296
1297 #
1298 # COMBINED FIGURES
1299 #
1300
1301 def plot_rl_diff_combined(results: Dict, out_dir: str):
1302     """Figure 19 - RL + Diffusion combined performance heatmap."""
1303     from config.base_config import SCENARIOS, RL_ALGOS, DIFF_ALGOS
1304     _setup_dark_style()

```

```

1300
1301     fig, axes = plt.subplots(1, 3, figsize=(18, 6))
1302     fig.suptitle('RL × Diffusion Combined Performance\n(task completion ×
        trajectory quality)',
1303                 fontweight='bold')
1304
1305     for ax, sc in zip(axes, SCENARIOS):
1306         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
1307         ax.set_xlabel('RL Algorithm')
1308         ax.set_ylabel('Diffusion Model')
1309
1310         mat = np.zeros((len(DIFF_ALGOS), len(RL_ALGOS)))
1311         for ri, rl in enumerate(RL_ALGOS):
1312             for di, diff in enumerate(DIFF_ALGOS):
1313                 # Combined score: RL completion × (1 / (1 + diff_loss))
1314                 rl_key = f'RL_{rl}_{sc}_s42'
1315                 diff_key = f'DIFF_{diff}_{rl}_{sc}_s42'
1316                 rl_comp = results.get(rl_key,
1317                                     {}).get('final_completion', 0.0)
1318                 diff_loss = results.get(diff_key, {}).get('best_loss',
1319                 1.0) or 1.0
1320                 mat[di, ri] = rl_comp * (1 / (1 + diff_loss))
1321
1322         if mat.max() > 0:
1323             mat /= mat.max()
1324
1325         cmap = LinearSegmentedColormap.from_list('perf', [PANEL_BG,
1326                 '#1a3a6b', ACCENT])
1327         im = ax.imshow(mat, cmap=cmap, aspect='auto', vmin=0, vmax=1)
1328         ax.set_xticks(range(len(RL_ALGOS)))
1329         ax.set_xticklabels(RL_ALGOS, rotation=30, ha='right', fontsize=8)
1330         ax.set_yticks(range(len(DIFF_ALGOS)))
1331         ax.set_yticklabels(DIFF_ALGOS, fontsize=8)
1332         for i in range(len(DIFF_ALGOS)):

```

```

1330         for j in range(len(RL_ALGOS)):
1331             ax.text(j, i, f'{mat[i,j]:.2f}', ha='center', va='center',
1332                     fontsize=7, color='white')
1333         fig.colorbar(im, ax=ax, shrink=0.8, label='Combined Score')
1334
1335     plt.tight_layout()
1336     _save(fig, f'{out_dir}/combined/19_rl_diff_combined.png')
1337
1338
1339 def plot_summary_table(results: Dict, out_dir: str):
1340     """Figure 20 - Summary results table (for thesis table 1)."""
1341     from config.base_config import SCENARIOS, RL_ALGOS
1342     _setup_dark_style()
1343
1344     rows = []
1345     for sc in SCENARIOS:
1346         for algo in RL_ALGOS:
1347             comp_vals, reward_vals, make_vals = [], [], []
1348             for seed in [42, 0, 1]:
1349                 key = f'RL_{algo}_{sc}_s{seed}'
1350                 if key in results and results[key].get('rewards'):
1351                     comp_vals.append(results[key].get('final_completion',
1352                                                         0))
1353                     reward_vals.append(results[key].get('final_reward',
1354                                                         0))
1354
1355             make_vals.append(np.mean(results[key].get('makespans', [200])[-100:]))
1356
1357             rows.append({
1358                 'Scenario': sc.replace('_', ' ').title()[:12],
1359                 'Algorithm': algo,
1360                 'Completion':
1361                     f'{np.mean(comp_vals)*100:.1f}±{np.std(comp_vals)*100:.1f}%'
1362                     if comp_vals else 'N/A',
1363                 'Reward':

```

```

1360         f'{np.mean(reward_vals):.1f}±{np.std(reward_vals):.1f}'
1361         if reward_vals else 'N/A',
1362         'Makespan': f'{np.mean(make_vals):.0f}' if make_vals
1363     else 'N/A',
1364     })
1365
1366 if not rows:
1367     print(' No results data for summary table --- skipping')
1368     return
1369
1370 fig, ax = plt.subplots(figsize=(14, max(6, len(rows) * 0.4 + 2)))
1371 fig.suptitle('Summary Results Table (mean ± std, 3 seeds)',
1372             fontweight='bold')
1373 ax.axis('off')
1374
1375 col_labels = ['Scenario', 'Algorithm', 'Completion', 'Reward',
1376             'Makespan']
1377 cell_text = [[r[c] for c in col_labels] for r in rows]
1378
1379 # Color rows by scenario
1380 sc_row_colors = []
1381 for row in rows:
1382     sc_name = row['Scenario'].lower().replace(' ', '_')
1383     sc_full = next((s for s in SCENARIOS if
1384 s.startswith(sc_name[:5])), SCENARIOS[0])
1385     col = SCENARIO_COLORS.get(sc_full, ACCENT)
1386     sc_row_colors.append([col + '33' * len(col_labels)] # 20%
1387         opacity hex
1388
1389 tbl = ax.table(cellText=cell_text, colLabels=col_labels,
1390               cellLoc='center', loc='center',
1391               bbox=[0, 0, 1, 1])
1392
1393 tbl.auto_set_font_size(False)
1394 tbl.set_fontsize(9)

```

```

1388
1389     for (r, c), cell in tbl.get_celld().items():
1390         cell.set_facecolor(PANEL_BG if r > 0 else '#1a3a6b')
1391         cell.set_edgecolor(GRID_COLOR)
1392         cell.set_text_props(color=TEXT_COLOR)
1393
1394     _save(fig, f'{out_dir}/combined/20_summary_table.png')
1395
1396
1397 #
1398 # hybrid-vs-base comparison figures
1399 #
1400
1401 def plot_hybrid_vs_base_rl(results: Dict, out_dir: str):
1402     """Figure 21 - HybridRL vs base RL final completion, per scenario."""
1403     from config.base_config import SCENARIOS, RL_ALGOS
1404     _setup_dark_style()
1405
1406     base_algos = [a for a in RL_ALGOS if a != 'HybridRL']
1407     fig, axes = plt.subplots(1, 3, figsize=(15, 4.5))
1408     fig.suptitle('HybridRL vs base RL (final completion %)',
1409                 fontweight='bold')
1410
1411     for ax, sc in zip(axes, SCENARIOS):
1412         vals = []
1413         labels = []
1414         for a in base_algos + ['HybridRL']:
1415             r = results.get(f'RL_{a}_{sc}_s42')
1416             v = ((r.get('final_completion', 0) or 0) * 100
1417                 if r and r.get('status') == 'ok' else 0.0)
1418             vals.append(v)
1419             labels.append(a)
1420         colors = ['#888888'] * len(base_algos) + ['#e34a33'] # hybrid
1421                        highlighted

```

```

1421         ax.bar(labels, vals, color=colors, edgecolor='black',
1422                linewidth=0.7)
1423         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
1424         ax.set_ylabel('Completion %')
1425         ax.set_ylim(0, max(100, max(vals) * 1.15) if vals else 100)
1426         ax.grid(axis='y', alpha=0.3)
1427         for i, v in enumerate(vals):
1428             ax.text(i, v + 1, f'{v:.1f}', ha='center', fontsize=8)
1429
1430     plt.tight_layout()
1431     _save(fig, f'{out_dir}/rl/21_hybrid_vs_base_rl.png')
1432
1433 def plot_hybrid_vs_base_diff(results: Dict, out_dir: str):
1434     """Figure 22 - HybridDiffusion vs base diffusion best loss, per
1435     scenario.
1436
1437     All conditioned on DIFF_RL_PAIR (HybridRL). Lower is better.
1438     """
1439     from config.base_config import SCENARIOS, DIFF_ALGOS, DIFF_RL_PAIR
1440     _setup_dark_style()
1441
1442     base_diffs = [d for d in DIFF_ALGOS if d != 'HybridDiffusion']
1443     fig, axes = plt.subplots(1, 3, figsize=(15, 4.5))
1444     fig.suptitle(f'HybridDiffusion vs base diffusion on {DIFF_RL_PAIR} ',
1445                 '(best loss, lower is better)',
1446                 fontweight='bold')
1447
1448     for ax, sc in zip(axes, SCENARIOS):
1449         vals = []
1450         labels = []
1451         for d in base_diffs + ['HybridDiffusion']:
1452             r = results.get(f'DIFF_{d}_{DIFF_RL_PAIR}_{sc}_s42')
1453             v = (r.get('best_loss', float('nan'))

```



```

1453         if r and r.get('status') == 'ok' else float('nan'))
1454         v = float(v) if isinstance(v, (int, float)) and v !=
float('inf') else 0.0
1455         vals.append(v)
1456         labels.append(d)
1457         colors = ['#888888'] * len(base_diffs) + ['#43a2ca']    # hybrid
highlighted
1458         ax.bar(labels, vals, color=colors, edgecolor='black',
linewidth=0.7)
1459         ax.set_title(SCENARIO_LABELS[sc], fontsize=10)
1460         ax.set_ylabel('Best loss')
1461         ax.tick_params(axis='x', labelsz=7)
1462         ax.grid(axis='y', alpha=0.3)
1463         for i, v in enumerate(vals):
1464             ax.text(i, v, f'{v:.3f}', ha='center', va='bottom',
fontsize=7)
1465
1466     plt.tight_layout()
1467     _save(fig, f'{out_dir}/diffusion/22_hybrid_vs_base_diff.png')
1468
1469
1470 #
1471 # MAIN ENTRY POINT
1472 #
1473
1474 def generate_all_figures(results_dir: str = 'results',
1475                         figures_dir: str = 'results/figures') ->
List[str]:
1476     """
1477     Generate all publication-ready figures.
1478
1479     loads the new 4-stage pkls (stageA_rl / stageB_hybridrl /
1480     stageC_diff / stageD_hybriddiff). Keeps legacy.pkl paths for
1481     backward compatibility with result directories.

```

```

1482
1483 Returns list of generated figure paths.
1484 """
1485     # Load results - stage files first, then legacy pkls
1486     results: Dict[str, Any] = {}
1487     pkl_names = [
1488         'stageA_rl.pkl', 'stageB_hybridrl.pkl',
1489         'stageC_diff.pkl', 'stageD_hybriddiff.pkl',
1490         # legacy fallbacks
1491         'phase1_rl.pkl', 'phase2_diff.pkl', 'phase2_diffusion.pkl',
1492         'all_results.pkl',
1493     ]
1494     for pkl_name in pkl_names:
1495         pkl_path = os.path.join(results_dir, pkl_name)
1496         if os.path.exists(pkl_path):
1497             # no silent try/except wrapper. If a pkl is corrupt, we want
1498             # the error visible, not swallowed. Per-figure errors are
1499             still
1500             # caught in _run below so one bad figure doesn't kill the set.
1501             with open(pkl_path, 'rb') as f:
1502                 results.update(pickle.load(f))
1503                 print(f'  Loaded {pkl_path}')
1504
1505         if not results:
1506             print('  No results found - generating figures with synthetic
1507             data only')
1508
1509     for subdir in ['eda', 'rl', 'diffusion', 'combined']:
1510         os.makedirs(os.path.join(figures_dir, subdir), exist_ok=True)
1511
1512     generated: List[str] = []
1513     print(f'\nGenerating figures -> {figures_dir}')
1514
1515     def _run(fn, *args):

```

```

1514         try:
1515             fn(*args)
1516             generated.append(fn.__name__)
1517         except Exception as e:
1518             print(f'  WARN: {fn.__name__} failed: {e}')
1519
1520     # EDA
1521     _run(plot_map_layouts,          figures_dir)
1522     _run(plot_robot_params,        figures_dir)
1523     _run(plot_task_location_heatmaps, figures_dir)
1524     _run(plot_scenario_complexity,  figures_dir)
1525     _run(plot_population_distributions, figures_dir)
1526
1527     # RL
1528     _run(plot_learning_curves, results, figures_dir)
1529     _run(plot_completion_rates, results, figures_dir)
1530     _run(plot_algorithm_comparison, results, figures_dir)
1531     _run(plot_makespan_comparison, results, figures_dir)
1532     _run(plot_task_allocation_heatmap, results, figures_dir)
1533     _run(plot_task_dependency_graph, figures_dir)
1534     _run(plot_hybrid_bandit_selection, results, figures_dir)
1535     _run(plot_variable_robot_performance, results, figures_dir)
1536     _run(plot_hybrid_vs_base_rl, results, figures_dir)      # NEW
1537
1538     # Diffusion
1539     _run(plot_diffusion_loss_curves, results, figures_dir)
1540     _run(plot_trajectory_visualization, figures_dir)
1541     _run(plot_bspline_vs_waypoints, figures_dir)
1542     _run(plot_denoising_sequence, figures_dir)
1543     _run(plot_inter_robot_avoidance, figures_dir)
1544     _run(plot_goal_conditioning, figures_dir)
1545     _run(plot_hybrid_vs_base_diff, results, figures_dir)    # NEW
1546
1547     # Combined

```

```

1548     _run(plot_rl_diff_combined, results, figures_dir)
1549     _run(plot_summary_table, results, figures_dir)
1550
1551     print(f'\nGenerated {len(generated)} figures')
1552     return generated

```

Listing B.20: Visualisation routines that produce learning curves, completion-rate bar charts, cross-algorithm heatmaps and the HybridRL bandit selection profile.

```

1  """SAR Land Robot trainer --- default agent_mode for search_and_rescue.
2
3  Wraps the shared ‘‘train_rl_agent’’ loop with ground-robot-specific setup:
4  - Hard-obstacle terrain traversal (debris blocks movement)
5  - Ground field-of-view (circular LOS limited by debris)
6  - Action space: discrete directions + speed
7  - Reward: Dec-POMDP formulation with partial observability
8
9  Activated when ‘‘SCENARIO_CONFIGS[‘search_and_rescue’].agent_mode ==
10     ‘land_robot’’
11  (the default). The drone path uses ‘‘sar_drone_trainer.py’’.
12  """
13  from __future__ import annotations
14
15  from typing import Any, Dict, Optional
16
17  import torch
18  import torch.nn as nn
19
20  from config.base_config import SCENARIO_CONFIGS
21  from training.rl_trainer import train_rl_agent
22
23

```

```

24 def train_sar_land_robot(
25     agent: nn.Module,
26     env,
27     seed: int,
28     algo: str = 'HybridRL',
29     num_episodes: int = 5000,
30     device: torch.device = torch.device('cpu'),
31     checkpoint_dir: str = 'models',
32     log_every: int = 50,
33     teacher_models: Optional[Dict[str, nn.Module]] = None,
34 ) -> Dict[str, Any]:
35     """Train a ground rescue robot agent on the SAR scenario.
36
37     Uses the standard 'train_rl_agent' loop with
38     scenario='search_and_rescue'.
39
40     The environment must be configured with:
41     - Hard terrain obstacles (debris as impassable cells requiring
42       pathfinding)
43     - Partial observability: circular FOV clipped by terrain LOS
44     - Ground movement model: no aerial shortcuts
45
46     Args:
47         agent: RL agent model to train.
48         env: LSAR environment instance
49             (ImprovedTaskAllocationEnv with
50               scenario='search_and_rescue').
51         seed: Random seed.
52         algo: Algorithm name (default HybridRL).
53         num_episodes: Training episode count.
54         device: Compute device.
55         checkpoint_dir: Checkpoint output directory.
56         log_every: Logging interval in episodes.
57         teacher_models: Optional teacher models for KL distillation
58             (HybridRL only).

```

```

54
55     Returns:
56         Training result dict (same schema as train_rl_agent).
57     """
58     cfg = SCENARIO_CONFIGS['search_and_rescue']
59     assert cfg.agent_mode == 'land_robot', (
60         f'sar_land_robot_trainer called but
61         agent_mode={cfg.agent_mode!r}. '
62         'Set agent_mode="land_robot" in SCENARIO_CONFIGS or use
63         sar_drone_trainer.'
64     )
65     return train_rl_agent(
66         agent=agent,
67         env=env,
68         scenario='search_and_rescue',
69         algo=algo,
70         seed=seed,
71         num_episodes=num_episodes,
72         device=device,
73         checkpoint_dir=checkpoint_dir,
74         log_every=log_every,
75         teacher_models=teacher_models,
76     )

```

Listing B.21: Search-and-rescue land robot trainer: configures ground robot constraints and delegates to the shared `train_rl_agent` loop.

```

1  """SAR Drone trainer --- activated when agent_mode == 'drone'.
2
3  Handles aerial-specific constraints that differ from the land robot:
4  - Soft terrain costs (obstacles slow flight, not hard blocks)
5  - Aerial FOV: wider radius, no terrain LOS blocking
6  - Action space: throttle + heading (continuous-like, discretised)
7  - Battery/fuel consumption model instead of ground battery drain

```

```

8 - No-coverage-overlap penalty between drones
9
10 Activated when ‘‘SCENARIO_CONFIGS[‘search_and_rescue’].agent_mode ==
    ‘drone’‘‘.
11 Change the flag in config/base_config.py to switch modes.
12 """
13
14 from __future__ import annotations
15
16 from typing import Any, Dict, Optional
17
18 import torch
19 import torch.nn as nn
20
21 from config.base_config import SCENARIO_CONFIGS
22 from training.rl_trainer import train_rl_agent
23
24
25 def _apply_drone_env_patches(env) -> None:
26     """Apply drone-mode patches to the LSAR environment at runtime.
27
28     Patches applied (in-place, do not persist across resets unless env
29     stores them):
30     - ‘‘env.agent_mode = ‘drone’‘ --- movement model key used by task_env
31     - ‘‘env.hard_obstacles = False‘‘ --- terrain is a soft cost, not a
32       hard block
33     - ‘‘env.aerial_fov = True‘‘ --- removes terrain LOS mask from
34       observations
35     """
36     if hasattr(env, ‘agent_mode’):
37         env.agent_mode = ‘drone’
38     if hasattr(env, ‘hard_obstacles’):
39         env.hard_obstacles = False
40     if hasattr(env, ‘aerial_fov’):

```

```

38         env.aerial_fov = True
39
40
41 def train_sar_drone(
42     agent: nn.Module,
43     env,
44     seed: int,
45     algo: str = 'HybridRL',
46     num_episodes: int = 5000,
47     device: torch.device = torch.device('cpu'),
48     checkpoint_dir: str = 'models',
49     log_every: int = 50,
50     teacher_models: Optional[Dict[str, nn.Module]] = None,
51 ) -> Dict[str, Any]:
52     """Train a coverage drone agent on the SAR scenario.
53
54     Patches the shared LSAR environment for aerial behaviour before
55     training,
56     then delegates to the standard 'train_rl_agent' loop.
57
58     Drone-specific differences vs land robot:
59     - Obstacles reduce speed (soft cost), do not block movement
60     - Wider aerial FOV, no terrain LOS clipping
61     - Battery/fuel drain proportional to flight time, not terrain cost
62     - Coverage-overlap penalty between drones (incentivises spread)
63
64     Args:
65         agent:          RL agent model to train.
66         env:            LSAR environment instance
67                        (ImprovedTaskAllocationEnv with
68                        scenario='search_and_rescue').
69         seed:          Random seed.
70         algo:          Algorithm name (default HybridRL).
71         num_episodes:  Training episode count.

```



```

70         device:          Compute device.
71         checkpoint_dir: Checkpoint output directory.
72         log_every:       Logging interval in episodes.
73         teacher_models: Optional teacher models for KL distillation.
74
75     Returns:
76         Training result dict (same schema as train_rl_agent).
77     """
78     cfg = SCENARIO_CONFIGS['search_and_rescue']
79     assert cfg.agent_mode == 'drone', (
80         f'sar_drone_trainer called but agent_mode={cfg.agent_mode!r}. '
81         'Set agent_mode="drone" in SCENARIO_CONFIGS or use
82         sar_land_robot_trainer.'
83     )
84     _apply_drone_env_patches(env)
85     return train_rl_agent(
86         agent=agent,
87         env=env,
88         scenario='search_and_rescue',
89         algo=algo,
90         seed=seed,
91         num_episodes=num_episodes,
92         device=device,
93         checkpoint_dir=checkpoint_dir,
94         log_every=log_every,
95         teacher_models=teacher_models,
96     )

```

Listing B.22: Search-and-rescue drone trainer: patches the environment for aerial kinematics and delegates to the shared `train_rl_agent` loop.

Package: utils

```
1 # utils/__init__.py
```

Listing B.23: Package init for the utilities package.

```
1 """
2 utils/action_masking.py
3 =====
4 Invalid action masking - adapted from TA-RWARE (Krnjaic et al. ICRA 2023).
5
6 Key insight: before the policy softmax, set logits of invalid actions to
7     -inf.
8 Agents cannot waste steps choosing completed or out-of-range tasks.
9 This alone dramatically improves convergence in sparse reward settings.
10
11 Works with all 5 RL algorithms via a common interface.
12 """
13 import torch
14 import numpy as np
15 from config.base_config import (
16     COLUMN_HEIGHT, STUCK_THRESHOLD,
17     TASK_STATUS_AVAILABLE,
18 )
19 from typing import List, Optional, Dict
20
21 def build_action_mask(tasks: List[Dict],
22                      num_robots: int,
23                      max_tasks: int,
24                      flat_obs: Optional[np.ndarray] = None,
25                      flat_task_start: int = 300,
26                      flat_task_feats: int = 12) -> torch.Tensor:
27     """
28     Build a boolean mask of shape (num_robots * max_tasks + 1,).
29     True = action is INVALID (will be set to -inf before softmax).
```

```

30 False = action is valid.
31
32 Action encoding: action = robot_id * max_tasks + task_id
33 Last action (index = num_robots * max_tasks) = no-op (always valid).
34
35 Args:
36 tasks: list of task dicts with 'status' key
37 num_robots: number of active robots
38 max_tasks: maximum tasks in action space
39 flat_obs: optional 910-dim flat observation (faster check)
40 flat_task_start: offset in flat_obs for task block
41 flat_task_feats: features per task in flat_obs
42
43 Returns:
44 mask: (action_dim,) bool tensor, True = invalid
45 """
46     action_dim = num_robots * max_tasks + 1
47     mask = torch.ones(action_dim, dtype=torch.bool)
48
49     # No-op is always valid
50     mask[-1] = False
51
52     for robot_id in range(num_robots):
53         for task_id in range(min(max_tasks, len(tasks))):
54             action_idx = robot_id * max_tasks + task_id
55
56             # Check if task is still pending or assigned (valid to target)
57             if flat_obs is not None:
58                 off = flat_task_start + task_id * flat_task_feats
59                 if off + flat_task_feats < len(flat_obs):
60                     is_completed = float(flat_obs[off + 7]) > 0.5
61                     # uninitialised slot
62                     lx = float(flat_obs[off + 0])
63                     ly = float(flat_obs[off + 1])

```

```

64         is_empty = (lx == 0.0 and ly == 0.0)
65         if not is_completed and not is_empty:
66             mask[action_idx] = False    # valid
67             # else stays True = invalid
68         else:
69             status = tasks[task_id].get('status',
TASK_STATUS_AVAILABLE)
70             if status == TASK_STATUS_AVAILABLE:
71                 mask[action_idx] = False    # valid
72
73     return mask
74
75
76 def mask_logits(logits: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
77     """
78     Apply action mask to logits. Sets INVALID action logits to -1e8.
79
80     Args:
81     logits: (... , action_dim)
82     mask: (action_dim,) -- bool (True=invalid) OR float (0=invalid, 1=valid)
83
84     TorchScript constraints obeyed:
85     - No mask.bool -- use comparison instead
86     - No dtype in (tuple) -- use explicit == with or
87     - No tensor[..., bool] -- use additive penalty broadcast instead
88     """
89     if mask is None:
90         return logits
91
92     action_dim = logits.shape[-1]
93
94     # Normalise to 1-D tensor of length action_dim
95     m = mask.view(-1)[:action_dim]
96     if m.shape[0] < action_dim:

```

```

97         pad = torch.ones(action_dim - m.shape[0], dtype=m.dtype,
98                             device=m.device)
99
100         m = torch.cat([m, pad])
101
102         # Build penalty vector: -1e8 for invalid actions, 0 for valid.
103         # Then add it -- broadcasts cleanly over (... , action_dim) without any
104         # boolean-index or ellipsis-index syntax that TorchScript rejects.
105         if m.dtype == torch.float32 or m.dtype == torch.float16 or m.dtype ==
106             torch.float64:
107             penalty = (m < 0.5).to(dtype=logits.dtype) * -1e8    # float mask:
108             0=invalid
109         else:
110             penalty = (m != 0).to(dtype=logits.dtype) * -1e8    # bool/int
111             mask: 1=invalid
112
113         return logits + penalty
114
115 def masked_softmax(logits: torch.Tensor, mask: torch.Tensor) ->
116     torch.Tensor:
117     """Softmax with invalid actions zeroed out."""
118     masked_l = mask_logits(logits, mask)
119     probs = torch.softmax(masked_l, dim=-1)
120     # Zero out any NaN (all-invalid edge case)
121     probs = torch.nan_to_num(probs, nan=0.0)
122     return probs
123
124 def masked_argmax(logits: torch.Tensor, mask: torch.Tensor) ->
125     torch.Tensor:
126     """
127     Deterministic greedy action selection with masking.
128     Used at evaluation time (DCMRTA: argmax(logp.exp * ~mask)).
129     """
130     log_probs = torch.log_softmax(mask_logits(logits, mask), dim=-1)
131     return torch.argmax(log_probs * (~mask).float(), dim=-1)

```

```

125
126
127 class ActionMaskWrapper:
128     """
129     Wraps any RL agent's action selection to automatically apply masking.
130     Used uniformly across MAPPO, QMIX, MADDPG, DQN, HybridRL.
131     """
132
133     def __init__(self, max_tasks: int = 50, max_robots: int = 20):
134         self.max_tasks = max_tasks
135         self.max_robots = max_robots
136
137     def get_mask(self, tasks: List[Dict], num_robots: int,
138                 flat_obs: Optional[np.ndarray] = None) -> torch.Tensor:
139         return build_action_mask(tasks, num_robots, self.max_tasks,
140                                 flat_obs)
141
142     def apply(self, logits: torch.Tensor,
143              tasks: List[Dict],
144              num_robots: int,
145              flat_obs: Optional[np.ndarray] = None,
146              deterministic: bool = False) -> torch.Tensor:
147         """
148         Apply mask and sample or argmax.
149
150         Returns:
151         action: scalar tensor
152         """
153         mask = self.get_mask(tasks, num_robots, flat_obs)
154         mask = mask.to(logits.device)
155
156         if deterministic:
157             return masked_argmax(logits, mask)
158         else:

```

```

158         probs = masked_softmax(logits, mask)
159         dist = torch.distributions.Categorical(probs=probs)
160         return dist.sample()
161
162
163     #
164     # TA-RWARE improvements H + I + M: spatial specialisation + conflict-free
165     # masking
166     # From TA-RWARE warehouse.py compute_valid_action_masks
167     #
168     def warehouse_action_mask(n_tasks: int,
169                               task_states: np.ndarray,
170                               robot_assignments: np.ndarray,
171                               robot_carrying: np.ndarray,
172                               block_conflicts: bool = True) -> np.ndarray:
173         """
174         TA-RWARE-style action masking for warehouse scenario.
175
176         Rules (from compute_valid_action_masks):
177         - If robot is NOT carrying: can only assign to REQUESTED (incomplete)
178           tasks
179         - If robot IS carrying: can only assign to empty delivery zones
180         - block_conflicts=True: two robots cannot target the same task
181           simultaneously
182         - Robots in same spatial zone get priority over same tasks (rack_groups
183           logic)
184
185         Args:
186         n_tasks: number of tasks in this episode
187         task_states: (n_tasks,) 0=empty/done, 1=requested/pending
188         robot_assignments: (n_robots,) current task assignment (-1=none)
189         robot_carrying: (n_robots,) bool - robot has payload
190         block_conflicts: prevent two robots targeting same task

```

```

188
189 Returns:
190 valid_mask: (n_robots, n_tasks+1) bool - True = valid action
191 column 0 is no-op, columns 1..n_tasks are task IDs
192 """
193     n_robots = len(robot_assignments)
194     mask      = np.zeros((n_robots, n_tasks + 1), dtype=bool)
195     mask[:, 0] = True    # no-op always valid
196
197     requested = np.where(task_states == 1)[0]    # pending tasks
198     empty_dest = np.where(task_states == 0)[0]    # delivery/empty slots
199
200     # Currently targeted tasks (for conflict prevention)
201     active_targets = set(a for a in robot_assignments if a >= 0)
202
203     for ri in range(n_robots):
204         if robot_carrying[ri]:
205             valid_tasks = empty_dest    # carrying: go to delivery zone
206         else:
207             valid_tasks = requested    # not carrying: pick requested task
208
209         for ti in valid_tasks:
210             if block_conflicts and ti in active_targets and
211                 robot_assignments[ri] != ti:
212                 continue    # another robot already targeting this - skip
213                 mask[ri, ti + 1] = True    # +1 because col 0 is no-op
214
215     return mask
216
217 #
218 # TA-RWARE improvement K: column-height-aware stuck counter
219 # From TA-RWARE warehouse.py: StuckCounter class + resolve_stuck_agents
220 # Threshold 1 (replan): count > STUCK_THRESHOLD

```



```

221 # Threshold 2 (free): count > STUCK_THRESHOLD + column_height + 2
222 #
223
224 class StuckCounter:
225     """
226     Position-aware stuck counter.
227     Resets when position changes; increments each step at same position.
228
229     TA-RWARE resolve_stuck_agents logic:
230     - STUCK_THRESHOLD < count < STUCK_THRESHOLD + column_height + 2
231     replan: recompute A* path / reassign task
232     - count > STUCK_THRESHOLD + column_height + 2
233     free: mark robot as available, clear assignment
234     """
235
236     def __init__(self, position: tuple, column_height: int =
        COLUMN_HEIGHT):
237         self.position      = position
238         self.count         = 0
239         self.column_height = column_height
240         self._replan_thresh = STUCK_THRESHOLD
241         self._free_thresh  = STUCK_THRESHOLD + column_height + 2
242
243     def update(self, new_position: tuple) -> str:
244         """
245         Update position. Returns 'ok'|'replan'|'free' status.
246         """
247         if new_position == self.position:
248             self.count += 1
249         else:
250             self.count = 0
251             self.position = new_position
252
253         if self.count > self._free_thresh:

```

```

254         return 'free'
255     elif self.count > self._replan_thresh:
256         return 'replan'
257     return 'ok'
258
259     def reset(self, position: tuple = None):
260         self.count = 0
261         if position is not None:
262             self.position = position
263
264     @property
265     def is_stuck(self) -> bool:
266         return self.count > self._replan_thresh
267
268
269     def resolve_stuck_robots(robots: list,
270                             stuck_counters: dict,
271                             column_height: int = COLUMN_HEIGHT) -> dict:
272         """
273         TA-RWARE stuck resolution for AQUA robots.
274
275         Args:
276         robots: list of robot dicts with 'id' and 'position' keys
277         stuck_counters: dict of {robot_id: StuckCounter}
278         column_height: shelf column height (default COLUMN_HEIGHT)
279
280         Returns:
281         dict {robot_id: 'ok'|'replan'|'free'}
282         """
283         statuses = {}
284         for robot in robots:
285             rid = robot.get('id', 0)
286             pos = tuple(robot.get('position', [0, 0]))
287

```

```

288         if rid not in stuck_counters:
289             stuck_counters[rid] = StuckCounter(pos, column_height)
290
291         status = stuck_counters[rid].update(pos)
292         statuses[rid] = status
293
294         if status == 'free':
295             # Clear assignment - robot was stuck too long to be useful
296             here
297             robot['assigned'] = False
298             robot['task_id'] = None
299             stuck_counters[rid].reset(pos)
300
301         return statuses
302
303     #
304     # Scenario-aware action mask factory (all 3 scenarios)
305     # Warehouse: TA-RWARE style (carryingdelivery, not-carryingrequested,
306     block conflicts)
307     # SAR: Victim priority mask - high-priority victims first, avoid
308     double-assignment
309     # Maritime: Sector mask - robots in same sector share tasks, distant
310     robots excluded
311     #
312
313     def get_scenario_action_mask(scenario: str,
314                                n_tasks: int,
315                                task_states: np.ndarray,
316                                robot_assignments: np.ndarray,
317                                robot_positions: np.ndarray,
318                                task_positions: np.ndarray,
319                                robot_carrying: np.ndarray = None,
320                                task_priorities: np.ndarray = None,

```

```

318             obs_radius_m: float = 15.0,
319             comm_range_m: float = 30.0,
320             block_conflicts: bool = True) -> np.ndarray:
321     """
322     Unified scenario-aware action masking for all 3 AQUA scenarios.
323
324     Warehouse (TA-RWARE H+I+M):
325     - carrying robots delivery zones only
326     - non-carrying requested tasks only
327     - block conflicting assignments
328
329     Search & Rescue (SAR priority masking):
330     - Always mask already-completed tasks
331     - Prefer high-priority victims (mask low-priority if high exists within
332       range)
333     - Block double-assignment (TA-RWARE M applied to SAR)
334
335     Maritime Security (sector masking):
336     - Robots can only be assigned tasks within their observable range
337     - Block double-assignment
338     - Prioritise tasks with higher threat level
339
340     Args:
341     scenario: 'warehouse' | 'search_and_rescue' | 'maritime_security'
342     n_tasks: number of tasks
343     task_states: (n_tasks,) 0=done, 1=pending, 2=in_progress
344     robot_assignments: (n_robots,) current task (-1=none)
345     robot_positions: (n_robots, 2) normalised positions
346     task_positions: (n_tasks, 2) normalised task locations
347     robot_carrying: (n_robots,) bool [warehouse only]
348     task_priorities: (n_tasks,) int 1-5 [SAR/Maritime]
349     block_conflicts: prevent two robots targeting same task
350
351     Returns:

```

```

351 valid_mask: (n_robots, n_tasks+1) bool - col 0 = no-op
352 """
353     n_robots = len(robot_assignments)
354     mask = np.zeros((n_robots, n_tasks + 1), dtype=bool)
355     mask[:, 0] = True    # no-op always valid
356
357     active_targets = set(a for a in robot_assignments if a >= 0)
358
359     if scenario == 'warehouse':
360         # Warehouse is RL-only mode (auto_reassign_on_complete=False):
361         the RL
362         # is the sole arbiter of all assignments. Only IDLE robots
363         (current_task
364         # == -1) may receive new assignments; masking out busy robots
365         prevents
366         # the policy from selecting actions that step() rejects with
367         PENALTY_INCOMPLETE,
368         # which would otherwise dominate the reward signal and kill
369         learning.
370         pending = np.where(task_states == 1)[0]
371         for ri in range(n_robots):
372             if robot_assignments[ri] >= 0:
373                 continue # robot busy --- no valid assignment until it
374                 finishes
375                 for ti in pending:
376                     if block_conflicts and ti in active_targets and
377                     robot_assignments[ri] != ti:
378                         continue
379                     mask[ri, ti + 1] = True
380
381     elif scenario == 'search_and_rescue':
382         # SAR: priority-aware masking adapted from TA-RWARE delivery logic
383         # High-priority victims (priority>=4) block assignment to
384         low-priority ones

```

```

377         # when a high-priority victim is within the robot's observable
378         range
379         if task_priorities is None:
380             task_priorities = np.ones(n_tasks, dtype=int)
381             pending = np.where(task_states == 1)[0]
382
383         for ri in range(n_robots):
384             if len(pending) == 0:
385                 break
386             rpos = robot_positions[ri] if len(robot_positions) > ri else
387             np.zeros(2)
388             # Compute distances to all pending tasks (raw metres)
389             dists = np.linalg.norm(task_positions[pending] - rpos, axis=1)
390             in_range = pending[dists < obs_radius_m]
391             max_prio_in_range = (task_priorities[in_range].max()
392                                 if len(in_range) > 0 else 1)
393
394             for j, ti in enumerate(pending):
395                 if block_conflicts and ti in active_targets and
396                 robot_assignments[ri] != ti:
397                     continue
398                 # Only allow low-priority tasks if no high-priority ones
399                 are nearby
400                 if (task_priorities[ti] >= max_prio_in_range - 1 or
401                     max_prio_in_range <= 2):
402                     mask[ri, ti + 1] = True
403
404     elif scenario == 'maritime_security':
405         # Maritime: range-based masking (raw metres via comm_range_m)
406         pending = np.where(task_states == 1)[0]
407
408         for ri in range(n_robots):
409             if len(pending) == 0:
410                 break

```

```

407         rpos = robot_positions[ri] if len(robot_positions) > ri else
np.zeros(2)
408         for ti in pending:
409             if block_conflicts and ti in active_targets and
robot_assignments[ri] != ti:
410                 continue
411                 dist = np.linalg.norm(task_positions[ti] - rpos)
412                 prio = task_priorities[ti] if task_priorities is not None
else 1
413                 if dist <= comm_range_m or prio >= 5:
414                     mask[ri, ti + 1] = True
415
416     return mask

```

Listing B.24: Action masking: validity rules and the additive logit penalty applied before the action softmax.

```

1  """
2  utils/bspline.py
3  =====
4  B-spline trajectory utilities.
5  Extracted from torch_robotics (MPD-splines deps) - no external install
    required.
6
7  Key idea (MPD-splines paper):
8  Instead of predicting N dense waypoints, predict K=8 control points of a
9  cubic B-spline. Evaluate the spline at T query times to get the
    trajectory.
10 This reduces output dimension from N*2 to K*2 and guarantees smoothness.
11
12 References:
13 Carvalho et al. "Motion Planning Diffusion" T-R0 2025
14 github.com/joaoamcarvalho/mpd-splines-public
15 """

```

```

16 import torch
17 import torch.nn.functional as F
18 import numpy as np
19 from typing import Optional
20
21
22 def bspline_basis(n_ctrl: int, degree: int, n_eval: int,
23                  device: torch.device) -> torch.Tensor:
24     """
25     Compute the B-spline basis matrix B of shape (n_eval, n_ctrl).
26     Multiplying control points P (n_ctrl, 2) by B^T gives trajectory
27         (n_eval, 2).
28
29     Uses a clamped uniform knot vector (curve passes through endpoints).
30
31     Args:
32     n_ctrl: number of control points (e.g. 8)
33     degree: spline degree (5 = quintic, per MPD-splines T-R02025)
34     n_eval: number of evaluation points (e.g. 20)
35     device: torch device
36
37     Returns:
38     B: (n_eval, n_ctrl) float32 tensor
39     """
40     # Clamped uniform knot vector
41     n_knots = n_ctrl + degree + 1
42     knots = np.zeros(n_knots)
43     interior = n_knots - 2 * (degree + 1)
44     knots[:degree + 1] = 0.0
45     if interior > 0:
46         knots[degree + 1:degree + 1 + interior] = np.linspace(0, 1,
47             interior + 2)[1:-1]
48     knots[-(degree + 1):] = 1.0

```



```

48     # Parameter values
49     t_vals = np.linspace(0.0, 1.0 - 1e-9, n_eval)
50
51     # Cox-de Boor recursion
52     B = np.zeros((n_eval, n_ctrl))
53     for j in range(n_ctrl):
54         # degree 0
55         b = np.zeros((n_knots - 1,))
56         for i in range(n_knots - 1):
57             b[i] = float(knots[i] <= t_vals[0] < knots[i + 1]) if i <
len(b) else 0.0
58
59     # Vectorised Cox-de Boor
60     # B[k, j] = N_{j,degree}(t_vals[k])
61     N = np.zeros((n_eval, n_knots - 1, degree + 1))
62     for i in range(n_knots - 1):
63         N[:, i, 0] = ((knots[i] <= t_vals) & (t_vals < knots[i +
1])).astype(float)
64     # Handle boundary: last eval point at t=1 belongs to last span
65     last_span = np.searchsorted(knots, 1.0, side='right') - 2
66     last_span = max(0, min(last_span, n_knots - 2))
67     N[-1, :, 0] = 0.0
68     N[-1, last_span, 0] = 1.0
69
70     for d in range(1, degree + 1):
71         for i in range(n_knots - 1 - d):
72             denom1 = knots[i + d] - knots[i]
73             denom2 = knots[i + d + 1] - knots[i + 1]
74             term1 = ((t_vals - knots[i]) / denom1 * N[:, i, d - 1]
75                     if denom1 > 1e-9 else np.zeros(n_eval))
76             term2 = ((knots[i + d + 1] - t_vals) / denom2 * N[:, i + 1,
d - 1]
77                     if denom2 > 1e-9 else np.zeros(n_eval))
78             N[:, i, d] = term1 + term2

```

```

79
80     B = N[:, :n_ctrl, degree]    # (n_eval, n_ctrl)
81     return torch.tensor(B, dtype=torch.float32, device=device)
82
83
84 class BSplineLayer(torch.nn.Module):
85     """
86     Differentiable B-spline evaluation layer.
87
88     forward(ctrl_pts) trajectory
89     ctrl_pts: (... , n_ctrl, 2) - predicted control points
90     trajectory: (... , n_eval, 2) - smooth trajectory waypoints
91
92     The basis matrix is precomputed once in __init__ (numpy is fine there -
93     TorchScript never compiles __init__) and stored via register_buffer so
94     that model.to(device) / model.cuda moves it automatically. This makes
95     forward pure-tensor and fully TorchScript-compatible.
96     """
97
98     # Explicit class-level annotation so TorchScript sees _B as Tensor,
99     not
100    # Optional[Tensor]. register_buffer guarantees it is always set.
101    _B: torch.Tensor
102
103    def __init__(self, n_ctrl: int = 8, degree: int = 5, n_eval: int = 20,
104                 zero_vel_bc: bool = True, zero_acc_bc: bool = True):
105        super().__init__()
106        self.n_ctrl = n_ctrl
107        self.degree = degree
108        self.zero_vel_bc = zero_vel_bc
109        self.zero_acc_bc = zero_acc_bc
110        self.n_eval = n_eval
111
112        # Compute on CPU; register_buffer moves it with the model to any
113        device.

```

```

111         # numpy is called here - __init__ is never traced or compiled by
        TorchScript.
112         B = bspline_basis(n_ctrl, degree, n_eval, torch.device('cpu'))
113         self.register_buffer('_B', B)    # shape: (n_eval, n_ctrl), float32
114
115     def _get_basis(self, device: torch.device) -> torch.Tensor:
116         """Return the precomputed basis matrix on *device*.
117
118         Called directly by diffusion_models.py for the least-squares
119         ctrl-point fitting step (outside of TorchScript). The matrix is
120         already a registered buffer so it normally lives on the same device
121         as the model; the explicit.to is a cheap no-op when devices match.
122         """
123         return self._B.to(device)
124
125     def preprocess_ctrl_pts(self, ctrl_pts: torch.Tensor) -> torch.Tensor:
126         """
127         Enforce zero-velocity and zero-acceleration boundary conditions.
128         MPD-splines T-R02025: ctrl[1]=ctrl[0], ctrl[-2]=ctrl[-1] (zero vel)
129         ctrl[2]=ctrl[0], ctrl[-3]=ctrl[-1] (zero acc)
130         Requires n_ctrl >= 6 for both BCs to be applied without overlap.
131         """
132         cp = ctrl_pts.clone()
133         if self.zero_vel_bc and self.n_ctrl >= 4:
134             cp[..., 1, :] = cp[..., 0, :]
135             cp[..., -2, :] = cp[..., -1, :]
136         if self.zero_acc_bc and self.n_ctrl >= 6:
137             cp[..., 2, :] = cp[..., 0, :]
138             cp[..., -3, :] = cp[..., -1, :]
139         return cp
140
141     def forward(self, ctrl_pts: torch.Tensor) -> torch.Tensor:
142         """
143         ctrl_pts: (... , n_ctrl, 2)

```

```

144 Returns (... , n_eval, 2) with zero-vel/acc BCs enforced (MPD-splines).
145 """
146         ctrl_pts = self.preprocess_ctrl_pts(ctrl_pts)
147         # self._B follows the model's device automatically via
           register_buffer.
148         # (... , n_eval, 2) = (... , n_eval, n_ctrl) @ (... , n_ctrl, 2)
149         return torch.matmul(self._B, ctrl_pts)
150
151
152 def smoothness_loss(traj: torch.Tensor) -> torch.Tensor:
153     """
154     Trajectory smoothness regularisation: penalise second-order differences.
155
156     Args:
157     traj: (batch, n_eval, 2)
158
159     Returns:
160     scalar loss
161     """
162     d2 = traj[:, 2:, :] - 2 * traj[:, 1:-1, :] + traj[:, :-2, :]
163     return (d2 ** 2).mean()
164
165 def inter_robot_collision_cost(trajs: torch.Tensor,
166                               radius: float = 0.5) -> torch.Tensor:
167     """
168     MMD spatio-temporal inter-robot collision cost.
169
170     During the diffusion denoising process, push robot trajectories apart
171     if they come within radius of each other at any timestep.
172
173     Args:
174     trajs: (n_robots, n_eval, 2) - trajectories for all robots
175     radius: collision radius in normalised units
176

```

```

177 Returns:
178 cost: scalar - sum of pairwise collision costs
179 """
180     n_robots, n_eval, _ = trajs.shape
181     if n_robots < 2:
182         return trajs.new_zeros(1).squeeze()
183
184     cost = trajs.new_zeros(1).squeeze()
185     for i in range(n_robots):
186         for j in range(i + 1, n_robots):
187             diff = trajs[i] - trajs[j]          # (n_eval, 2)
188             dist = torch.norm(diff, dim=-1)      # (n_eval,)
189             violation = torch.clamp(radius - dist, min=0.0)
190             cost = cost + (violation ** 2).sum()
191     return cost
192
193
194 def inter_robot_collision_gradient(trajs: torch.Tensor,
195                                   radius: float = 0.5
196                                   ) -> torch.Tensor:
197     """
198     MMD: gradient of inter-robot collision cost w.r.t. trajectories.
199     Used to guide the diffusion denoising step.
200
201     Args:
202     trajs: (n_robots, n_eval, 2)
203     radius: collision radius
204
205     Returns:
206     grad: (n_robots, n_eval, 2) - gradient to subtract from noisy
207           trajectories
208     """
209     n_robots, n_eval, _ = trajs.shape
210     grad = torch.zeros_like(trajs)

```

```

210     for i in range(n_robots):
211         for j in range(i + 1, n_robots):
212             diff = trajs[i] - trajs[j]                # (n_eval, 2)
213             dist = torch.norm(diff, dim=-1, keepdim=True).clamp(min=1e-6)
214             violation = torch.clamp(radius - dist, min=0.0) # (n_eval, 1)
215             # gradient pushes robot i away from robot j
216             g = 2.0 * violation * (-diff / (dist ** 2 + 1e-8))
217             grad[i] -= g
218             grad[j] += g
219     return grad

```

Listing B.25: B-spline trajectory representation and inter-robot collision constraint gradient (MMD-derived).

```

1  """
2  utils/gpu_utils.py
3  =====
4  GPU memory reservation for shared-GPU training.
5
6  Strategy: burst-allocate release to PyTorch's internal cache neutralise
7  empty_cache for the process lifetime.
8
9  Why this works
10 -----
11 PyTorch has a caching allocator. When you 'del' a CUDA tensor, the memory
    does
12 NOT go back to the driver; it sits in PyTorch's internal cache.
    'nvidia-smi'
13 still reports it as "used by this process" because the driver gave those
    pages
14 to us and has no knowledge of PyTorch's internal cache state.
15
16 Consequence: if we allocate a 14 GB block with torch.zeros, delete it, and
17 NEVER call torch.cuda.empty_cache, those 14 GB stay allocated to our

```

```

    process
18 from the driver's perspective (blocking other users), while being
    available to
19 training tensors through PyTorch's allocator. One pool, two purposes.
20
21 The one catch is that any torch.cuda.empty_cache call would undo the
    reservation
22 by flushing the cache back to the driver. AQUA's code has ~12 such call
    sites.
23 We neutralise them process-wide via a monkey-patch once the reservation
    is active.
24
25 Usage (from a worker):
26 from utils.gpu_utils import setup_gpu_memory
27 setup_gpu_memory # reads AQUA_GPU_GB_PER_WORKER env var
28 # nvidia-smi now shows ~(cap * 0.95) GB used by this PID
29 # Training continues normally - allocations reuse the reserved memory
30 """
31 from __future__ import annotations
32 import os
33 import time
34 from typing import Dict, List, Optional
35
36 # Module-level state. These survive for the process lifetime.
37 _GPU_RESERVED_GB: float = 0.0
38 _ORIGINAL_EMPTY_CACHE = None # original torch.cuda.empty_cache,
    restored on release
39
40
41 def query_free_gpu_gb(device_index: int = 0) -> Dict[str, float]:
42     """Return {'free_gb', 'total_gb', 'used_gb'} or zeros if no CUDA."""
43     try:
44         import torch
45         if not torch.cuda.is_available():

```

```

46         return {'free_gb': 0.0, 'total_gb': 0.0, 'used_gb': 0.0}
47     free, total = torch.cuda.mem_get_info(device_index)
48     gb = 1024 ** 3
49     return {
50         'free_gb': free / gb,
51         'total_gb': total / gb,
52         'used_gb': (total - free) / gb,
53     }
54 except Exception:
55     return {'free_gb': 0.0, 'total_gb': 0.0, 'used_gb': 0.0}
56
57
58 def compute_worker_budget(
59     num_workers: int,
60     reserve_gb: Optional[float] = None,
61     headroom_gb: float = 2.0,
62     min_per_worker_gb: float = 2.0,
63 ) -> Dict[str, float]:
64     """
65     Compute per-worker GB budget from free VRAM + intended worker count.
66     Reduces worker count if per-worker would fall below min_per_worker_gb.
67     """
68     info = query_free_gpu_gb()
69     total_gb = info['total_gb']
70     free_gb = info['free_gb']
71
72     if total_gb <= 0:
73         return {
74             'num_workers': num_workers,
75             'per_worker_gb': 0.0,
76             'total_budget_gb': 0.0,
77             'free_gb': 0.0,
78             'total_gb': 0.0,
79             'fraction': 0.0,

```



```

80     }
81
82     if reserve_gb is None:
83         total_budget_gb = max(min_per_worker_gb, free_gb - headroom_gb)
84     else:
85         total_budget_gb = min(reserve_gb, free_gb - headroom_gb)
86         total_budget_gb = max(total_budget_gb, min_per_worker_gb)
87
88     n = max(1, int(num_workers))
89     while n > 1 and (total_budget_gb / n) < min_per_worker_gb:
90         n -= 1
91
92     per_worker_gb = total_budget_gb / n
93     fraction = per_worker_gb / total_gb
94
95     return {
96         'num_workers':    n,
97         'per_worker_gb':  per_worker_gb,
98         'total_budget_gb': total_budget_gb,
99         'free_gb':        free_gb,
100        'total_gb':        total_gb,
101        'fraction':        fraction,
102    }
103
104
105 def _noop_empty_cache() -> None:
106     """Replacement for torch.cuda.empty_cache when a reservation is
107     active."""
108     # Intentional no-op. Flushing the cache would release our reservation.
109     pass
110
111 def setup_gpu_memory(
112     per_worker_gb: Optional[float] = None,

```

```

113     device_index: int = 0,
114     prewarm: bool = True,
115 ) -> Dict[str, float]:
116     """
117     Reserve 'per_worker_gb' of VRAM for this process.
118
119     Flow:
120     1. Set a hard process cap via set_per_process_memory_fraction.
121     2. Burst-allocate 95% of the cap in 1-GB chunks with smaller fallbacks
122     (handles fragmentation / other users' residual allocations).
123     3. 'del' the chunks - memory goes to PyTorch's caching allocator, NOT
124     back to the driver, so nvidia-smi still shows it held by this PID.
125     4. Monkey-patch torch.cuda.empty_cache to a no-op so nothing in the
126     training loop can flush our reservation back to the driver.
127
128     After this returns:
129     - nvidia-smi shows  $\sim(\text{per\_worker\_gb} \times 0.95)$  GB used by this PID
130     - Training can freely allocate up to per_worker_gb - allocations reuse
131     the reserved memory via PyTorch's cache
132     - Other users cannot grab this memory until the process exits
133
134     Reads env vars if args are None:
135     AQUA_GPU_GB_PER_WORKER: per-worker cap (float GB)
136     AQUA_GPU_LOCK_PCT: fraction of cap to physically claim, default 0.95
137     """
138     global _GPU_RESERVED_GB, _ORIGINAL_EMPTY_CACHE
139
140     try:
141         import torch
142         if not torch.cuda.is_available():
143             return {}
144
145         # Idempotency
146         if _GPU_RESERVED_GB > 0:

```

```

147         total_gb =
torch.cuda.get_device_properties(device_index).total_memory / 1024 **
3
148     return {
149         'reserved_gb':     _GPU_RESERVED_GB,
150         'cap_gb':         per_worker_gb or _GPU_RESERVED_GB,
151         'total_gb':       total_gb,
152         'already_reserved': True,
153     }
154
155     # Resolve budget from env if not passed
156     if per_worker_gb is None:
157         v = os.environ.get('AQUA_GPU_GB_PER_WORKER', '').strip()
158         if v:
159             try:
160                 per_worker_gb = float(v)
161             except ValueError:
162                 per_worker_gb = None
163
164     if per_worker_gb is None or per_worker_gb <= 0:
165         info = query_free_gpu_gb(device_index)
166         return {'reserved_gb': 0.0, **info}
167
168     total_gb =
torch.cuda.get_device_properties(device_index).total_memory / 1024 **
3
169
170     # Hard cap - PyTorch will refuse allocations beyond this
171     fraction = min(1.0, per_worker_gb / total_gb)
172     try:
173         torch.cuda.set_per_process_memory_fraction(float(fraction),
device_index)
174     except Exception:
175         pass

```

```

176
177     # Fraction of cap to physically claim. 0.95 leaves a small cushion
178     # for PyTorch allocator overhead (CUDA context, kernel launches).
179     try:
180         lock_pct = float(os.environ.get('AQUA_GPU_LOCK_PCT', '0.95'))
181     except ValueError:
182         lock_pct = 0.95
183     lock_pct = max(0.50, min(0.99, lock_pct))
184     target_gb = per_worker_gb * lock_pct
185
186     # Burst allocation
187     chunks: List = []
188     claimed_gb = 0.0
189     chunk_gb = min(1.0, target_gb)
190     t0 = time.time()
191
192     if prewarm:
193         print(f' [gpu pid={os.getpid()}] Reserving {target_gb:.2f}
GB ',
194             f'(cap {per_worker_gb:.2f} GB, total {total_gb:.1f}
GB)...',
195             flush=True)
196
197     while claimed_gb < target_gb - 0.05:
198         need_gb = min(chunk_gb, target_gb - claimed_gb)
199         try:
200             elems = max(1, int(need_gb * 1024 ** 3 / 4))
201             # torch.zeros touches every page - forces physical
mapping
202             c = torch.zeros(
203                 elems, dtype=torch.float32,
204                 device=f'cuda:{device_index}',
205             )
206             chunks.append(c)

```

```

207         claimed_gb += need_gb
208     except Exception:
209         # Fragmentation or contention - try smaller chunks
210         chunk_gb /= 2.0
211         if chunk_gb < 0.05:
212             break
213
214     elapsed = time.time() - t0
215     print(f' [gpu pid={os.getpid()}] Reserved {claimed_gb:.2f}
GB ',
216         f'in {len(chunks)} chunks in {elapsed:.1f}s',
217         flush=True)
218
219     _GPU_RESERVED_GB = claimed_gb
220
221     # Release chunks to PyTorch's cache
222     # NOT to the driver. nvidia-smi will still show this memory as
used
223     # by our PID. Training tensors will reuse it via the cache.
224     del chunks
225
226     # Neutralise empty_cache
227     # Any torch.cuda.empty_cache elsewhere in this process would flush
228     # our cache back to the driver and lose the reservation. Replace
with
229     # a no-op. Restored by release_gpu_reservation at shutdown.
230     if _ORIGINAL_EMPTY_CACHE is None:
231         _ORIGINAL_EMPTY_CACHE = torch.cuda.empty_cache
232         torch.cuda.empty_cache = _noop_empty_cache
233
234     # Auto-shrink batch sizes on small budgets (< 4 GB per worker)
235     # Skip when run_fast.py (or the user) explicitly set
AQUA_NO_AUTO_SHRINK=1
236     if per_worker_gb < 4.0 and os.environ.get('AQUA_NO_AUTO_SHRINK',

```

```

'0') != '1':
237         try:
238             from config.base_config import RLConfig, DiffusionConfig
239             RLConfig.batch_size          = max(32, RLConfig.batch_size
// 2)
240             DiffusionConfig.batch_size = max(16,
DiffusionConfig.batch_size // 2)
241         except Exception:
242             pass
243
244         return {
245             'reserved_gb':          claimed_gb,
246             'cap_gb':              per_worker_gb,
247             'fraction':            fraction,
248             'total_gb':            total_gb,
249             'lock_pct':            lock_pct,
250             'empty_cache_suppressed': True,
251         }
252
253     except Exception as e:
254         import traceback
255         print(f' [gpu pid={os.getpid()}] setup_gpu_memory failed: {e}\n'
256               f'{traceback.format_exc()[:400]}',
257               flush=True)
258         return {'error': str(e)}
259
260
261 def release_gpu_reservation() -> None:
262     """
263     Release the reservation and restore torch.cuda.empty_cache.
264     For process shutdown or test cleanup only. Workers in normal operation
265     should NOT call this.
266     """
267     global _GPU_RESERVED_GB, _ORIGINAL_EMPTY_CACHE

```

```

268     _GPU_RESERVED_GB = 0.0
269     try:
270         import torch
271         if _ORIGINAL_EMPTY_CACHE is not None:
272             torch.cuda.empty_cache = _ORIGINAL_EMPTY_CACHE
273             _ORIGINAL_EMPTY_CACHE = None
274         if torch.cuda.is_available():
275             torch.cuda.empty_cache()
276     except Exception:
277         pass
278
279
280 # Backward-compat alias for old code that imported release_gpu_lock
281 release_gpu_lock = release_gpu_reservation

```

Listing B.26: GPU memory management utilities including the monkey-patched `torch.cuda.empty_cache` no-op for the parallel job runner.

```

1  """
2  utils/noise_schedule.py
3  =====
4  Cosine and linear noise schedules for diffusion models.
5  Extracted from MPD-splines (Carvalho et al. T-R0 2025).
6
7  Cosine schedule keeps signal-to-noise ratio higher for longer,
8  producing cleaner denoising for trajectory data.
9  """
10 import torch
11 import numpy as np
12 import math
13
14
15 def cosine_beta_schedule(T: int, s: float = 0.008) -> torch.Tensor:
16     """

```

```

17 Cosine noise schedule from Nichol & Dhariwal 2021.
18
19 Args:
20 T: total diffusion timesteps
21 s: offset to prevent  $\beta_0$  from being too small (default 0.008)
22
23 Returns:
24 betas: (T,) float32 tensor, values in (0, 1)
25 """
26     steps = T + 1
27     x = torch.linspace(0, T, steps)
28     alphas_cumprod = torch.cos(((x / T) + s) / (1 + s) * math.pi * 0.5)
29     ** 2
30     alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
31     betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
32     return torch.clamp(betas, 0.0001, 0.9999)
33
34 def linear_beta_schedule(T: int, beta_min: float = 1e-4,
35                           beta_max: float = 0.02) -> torch.Tensor:
36     """
37     Linear noise schedule (original DDPM).
38     """
39     return torch.linspace(beta_min, beta_max, T)
40
41
42 def get_schedule(schedule: str, T: int, **kwargs) -> torch.Tensor:
43     if schedule == 'cosine':
44         return cosine_beta_schedule(T, **kwargs)
45     elif schedule == 'linear':
46         return linear_beta_schedule(T, **kwargs)
47     else:
48         raise ValueError(f'Unknown schedule: {schedule}')
49

```



```

50
51 class DiffusionSchedule:
52     """
53     Precomputed diffusion schedule quantities.
54     Stores all values needed for forward process and denoising.
55     """
56
57     def __init__(self, schedule: str = 'cosine', T: int = 100,
58                 device: torch.device = torch.device('cpu')):
59         self.T = T
60         self.device = device
61
62         betas = get_schedule(schedule, T).to(device)
63         alphas = 1.0 - betas
64         alphas_cumprod = torch.cumprod(alphas, dim=0)
65         alphas_cumprod_prev = torch.cat([torch.ones(1, device=device),
66                                         alphas_cumprod[:-1]])
67
68         self.betas = betas
69         self.alphas = alphas
70         self.alphas_cumprod = alphas_cumprod
71         self.alphas_cumprod_prev = alphas_cumprod_prev
72         self.sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
73         self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1.0 -
74 alphas_cumprod)
75         self.log_one_minus_alphas_cumprod = torch.log(1.0 -
76 alphas_cumprod)
77         self.sqrt_recip_alphas_cumprod = torch.sqrt(1.0 /
78 alphas_cumprod)
79         self.sqrt_recipm1_alphas_cumprod = torch.sqrt(1.0 /
80 alphas_cumprod - 1)
81
82         # Posterior variance  $q(x_{t-1} | x_t, x_0)$ 
83         posterior_variance = (betas * (1.0 - alphas_cumprod_prev)
84                               / (1.0 - alphas_cumprod))

```

```

80         self.posterior_variance          = posterior_variance
81         self.posterior_log_var_clipped   = torch.log(
82             torch.clamp(posterior_variance, min=1e-20))
83         self.posterior_mean_coef1 = (betas *
torch.sqrt(alphas_cumprod_prev)
84                                     / (1.0 - alphas_cumprod))
85         self.posterior_mean_coef2 = ((1.0 - alphas_cumprod_prev)
86                                     * torch.sqrt(alphas)
87                                     / (1.0 - alphas_cumprod))
88
89     def q_sample(self, x0: torch.Tensor,
90                 t: torch.Tensor,
91                 noise: torch.Tensor = None) -> torch.Tensor:
92         """
93         Forward diffusion
94         """
95         if noise is None:
96             noise = torch.randn_like(x0)
97         sqrt_acp = self._extract(self.sqrt_alphas_cumprod, t, x0)
98         sqrt_omacp = self._extract(self.sqrt_one_minus_alphas_cumprod, t,
x0)
99         return sqrt_acp * x0 + sqrt_omacp * noise
100
101     def predict_x0_from_eps(self, x_t: torch.Tensor,
102                             t: torch.Tensor,
103                             eps: torch.Tensor) -> torch.Tensor:
104         """Recover x_0 estimate from predicted noise  $\epsilon$ ."""
105         recip = self._extract(self.sqrt_recip_alphas_cumprod, t, x_t)
106         recip1 = self._extract(self.sqrt_recipm1_alphas_cumprod, t, x_t)
107         return recip * x_t - recip1 * eps
108
109     def ddim_step(self, x_t: torch.Tensor, t: int, t_prev: int,
110                  eps: torch.Tensor) -> torch.Tensor:
111         """

```

```

112 DDIM deterministic sampling step.
113
114 Args:
115 x_t: (batch,...) noisy sample at step t
116 t, t_prev: scalar integers
117 eps: (batch,...) predicted noise
118
119 Returns:
120 x_{t-1}: (batch,...) denoised sample
121 """
122     acp_t = self.alphas_cumprod[t]
123     acp_prev = self.alphas_cumprod[t_prev] if t_prev >= 0 else
124         torch.tensor(1.0)
125
126     pred_x0 = (x_t - torch.sqrt(1 - acp_t) * eps) / torch.sqrt(acp_t)
127     pred_x0 = torch.clamp(pred_x0, -1.0, 1.0)
128
129     x_prev = (torch.sqrt(acp_prev) * pred_x0
130             + torch.sqrt(1 - acp_prev) * eps)
131
132     return x_prev
133
134 @staticmethod
135 def _extract(a: torch.Tensor, t: torch.Tensor,
136             x_shape: torch.Tensor) -> torch.Tensor:
137     """Extract values at timestep t and reshape for broadcasting."""
138     batch = t.shape[0]
139     out = a.gather(-1, t.long())
140     return out.reshape(batch, *((1,) * (len(x_shape.shape) - 1)))

```

Listing B.27: Cosine and linear noise schedule implementations for the diffusion forward process.

```

1 """
2 utils/oom_guard.py

```

```

3 =====
4 Catch CUDA OOM during a backward pass, halve the effective batch, retry.
5
6 Usage (in a trainer):
7     from utils.oom_guard import oom_safe
8
9     @oom_safe # decorator form - retries the whole function with halved batch
10     def _train_step(batch, model, opt, scaler):
11     ...
12
13 Or manual form inside a loop:
14
15     from utils.oom_guard import try_step
16
17     def step(batch):
18     ... one optimizer.step worth of work...
19
20     try_step(step, batch) # retries on OOM with halved batch
21
22 Opt-in via env var AQUA_OOM_RETRY=1 (set by run_fast.py). When not set,
23     both
24 helpers pass through untouched.
25 """
26 from __future__ import annotations
27 import os
28 from typing import Any, Callable
29
30 def _oom_enabled() -> bool:
31     return os.environ.get('AQUA_OOM_RETRY', '0') == '1'
32
33
34 def _is_cuda_oom(exc: BaseException) -> bool:
35     """Identify CUDA OOM across PyTorch versions."""

```

```

36     try:
37         import torch
38         if isinstance(exc, torch.cuda.OutOfMemoryError):
39             return True
40     except Exception:
41         pass
42     s = str(exc).lower()
43     return ('out of memory' in s) or ('cuda oom' in s) or ('cuda error:
44         out of memory' in s)
45
46 def _free_and_warn(tag: str, new_batch: int) -> None:
47     try:
48         import torch
49         # Normally empty_cache is a no-op when run_fast's reservation is
50         active,
51         # but calling it here doesn't hurt - worst case it's silently
52         skipped.
53         if torch.cuda.is_available():
54             torch.cuda.empty_cache()
55     except Exception:
56         pass
57     print(f' [oom_guard] {tag} OOM caught --- halving batch to
58         {new_batch}, retrying', flush=True)
59
60 def try_step(fn: Callable[..., Any], *args, tag: str = 'step',
61             max_halvings: int = 3, min_batch: int = 8,
62             batch_arg_index: int = 0, **kwargs) -> Any:
63     """
64     Call fn(*args, **kwargs). If the first positional argument looks like a
65     tensor and we hit CUDA OOM, halve its batch dimension and retry up to

```

```

66 This is a best-effort guard - it works when 'fn' is pure wrt its first
67 positional arg. For composite updates that read batched buffers from
68 elsewhere, use the config-level AQUA_RL_BATCH_SIZE / AQUA_DIFF_BATCH_SIZE
69 override instead.
70 """
71     if not _oom_enabled():
72         return fn(*args, **kwargs)
73
74     try:
75         return fn(*args, **kwargs)
76     except Exception as e:
77         if not _is_cuda_oom(e):
78             raise
79
80     # Halving retry loop
81     args = list(args)
82     for attempt in range(1, max_halvings + 1):
83         try:
84             t = args[batch_arg_index]
85             B = t.shape[0]
86             if B < min_batch * 2:
87                 raise RuntimeError(f'batch already at minimum ({B})')
88             new_B = B // 2
89             args[batch_arg_index] = t[:new_B]
90             _free_and_warn(tag, new_B)
91             return fn(*args, **kwargs)
92         except Exception as e:
93             if not _is_cuda_oom(e) or attempt == max_halvings:
94                 raise
95
96     # Unreachable
97     return None
98
99 def oom_safe(fn: Callable[..., Any]) -> Callable[..., Any]:

```

```

100     """Decorator form of try_step for functions whose first positional
101     arg is
102     the batched tensor."""
103     def wrapped(*args, **kwargs):
104         return try_step(fn, *args, tag=fn.__name__, **kwargs)
105     wrapped.__name__ = fn.__name__
106     wrapped.__doc__ = fn.__doc__
107     return wrapped
108
109 def shrink_config_batches_on_oom(fraction: float = 0.5) -> None:
110     """
111     Process-wide emergency: halve RLConfig.batch_size and
112     DiffusionConfig.batch_size.
113     Call this from a broad except clause if you hit repeated OOMs and want
114     all
115     subsequent iterations to use smaller batches.
116     """
117     try:
118         from config.base_config import RLConfig, DiffusionConfig
119         RLConfig.batch_size = max(16, int(RLConfig.batch_size
120         * fraction))
121         DiffusionConfig.batch_size = max(8,
122         int(DiffusionConfig.batch_size * fraction))
123         print(f' [oom_guard] globally shrunk batches -> '
124               f'RL={RLConfig.batch_size},
125               Diff={DiffusionConfig.batch_size}',
126               flush=True)
127     except Exception:
128         pass

```

Listing B.28: Out-of-memory recovery: catches CUDA OOM exceptions and retries the failed batch at a reduced batch size.

```

1  """
2  utils/reproducibility.py
3  =====
4  Seed management for reproducible multi-seed training.
5  """
6  import torch
7  import numpy as np
8  import random
9  import os
10
11
12  class ReproducibilityManager:
13      """Set all random seeds consistently across PyTorch, NumPy, Python
14      random."""
15
16      def __init__(self, seed: int):
17          self.seed = seed
18          self.apply(seed)
19
20      def apply(self, seed: int):
21          self.seed = seed
22          random.seed(seed)
23          np.random.seed(seed)
24          torch.manual_seed(seed)
25          torch.cuda.manual_seed_all(seed)
26          torch.backends.cudnn.deterministic = True
27          torch.backends.cudnn.benchmark = False
28          os.environ['PYTHONHASHSEED'] = str(seed)
29
30      def worker_seed(self, worker_id: int) -> int:
31          """Derive a unique but reproducible seed for a parallel worker."""
32          return self.seed + worker_id * 1000
33

```



```

34 def setup_cuda():
35     """Configure CUDA for shared GPU server training."""
36     os.environ.setdefault('PYTORCH_CUDA_ALLOC_CONF',
37         'expandable_segments:True')
38
39     if torch.cuda.is_available():
40         torch.cuda.empty_cache()
41         device = torch.device('cuda:0')
42         gpu_name = torch.cuda.get_device_name(0)
43         mem_total = torch.cuda.get_device_properties(0).total_memory / 1e9
44         print(f"GPU: {gpu_name} | VRAM: {mem_total:.1f} GB")
45     else:
46         device = torch.device('cpu')
47         print("No GPU found --- using CPU")
48
49     return device

```

Listing B.29: Reproducibility seed management: propagates a single seed through PyTorch, NumPy and the Python random module.

Package: parallel

```

1 # parallel/__init__.py

```

Listing B.30: Package init for the parallel package.

```

1 """
2 parallel/job_runner.py
3 =====
4 Multiprocessing training orchestrator used by run.py.
5
6 Schedules the 30 training jobs (5 RL × 3 scenarios + 5 diff × 3
   scenarios)

```

```

7 across worker processes with VRAM budgeting via AQUA_GPU_GB_PER_WORKER.
8
9 Prefers multiprocessing over Ray to avoid the extra Ray install dependency
10 on constrained cluster environments.
11
12 Usage (standalone, for debugging):
13     python parallel/job_runner.py          # full training
14     python parallel/job_runner.py --quick  # smoke test
15     python parallel/job_runner.py --rl_only # skip diffusion stages
16 """
17 import os
18 import sys
19 import time
20 import argparse
21 import logging
22 import pickle
23 import multiprocessing as mp
24 from pathlib import Path
25 from typing import Dict, Any, List, Tuple
26
27 import torch
28 import numpy as np
29
30 # Project imports
31 sys.path.insert(0, str(Path(__file__).parent.parent))
32
33 from config.base_config import (
34     SCENARIOS, RL_ALGOS, DIFF_ALGOS, SEEDS,
35     RESULTS_DIR, MODELS_DIR, SIM_READY_DIR, LOGS_DIR,
36     RLConfig, DiffusionConfig, ParallelConfig,
37     ROBOT_STATUS_ACTIVE,
38 )
39 from utils.reproducibility import setup_cuda, ReproducibilityManager
40 from training.statistical_eval import export_results_to_excel

```

```

41 from models.sim_ready_wrappers import export_sim_ready
42
43 rl_cfg = RLConfig()
44 diff_cfg = DiffusionConfig()
45 par_cfg = ParallelConfig()
46
47 # Logging
48
49 def setup_logging(log_dir: str = LOGS_DIR) -> logging.Logger:
50     os.makedirs(log_dir, exist_ok=True)
51     log_path = os.path.join(log_dir, 'aqua_v2_training.log')
52     logging.basicConfig(
53         level=logging.INFO,
54         format='%(asctime)s %(levelname)-8s %(message)s',
55         handlers=[logging.FileHandler(log_path), logging.StreamHandler()])
56     return logging.getLogger('aqua_v2')
57
58
59 # Worker functions (run in child processes)
60
61 def _rl_worker(job: Dict) -> Dict:
62     """Single RL training job - runs in a subprocess."""
63     import torch
64     os.environ.setdefault('PYTORCH_CUDA_ALLOC_CONF',
65         'expandable_segments:True')
66     torch.cuda.empty_cache()
67
68     algo = job['algo']
69     scenario = job['scenario']
70     seed = job['seed']
71     quick = job.get('quick', False)
72     num_ep = 5 if quick else rl_cfg.num_episodes
73
74     try:

```

```

74     from utils.reproducibility import ReproducibilityManager
75     from models.rl_agents import build_rl_agent
76     # train_rl_agent_with_real_env wrapper was deleted. Call
77     # train_rl_agent directly, same signature apart from the wrapper.
78     from training.rl_trainer import train_rl_agent
79
80     repro = ReproducibilityManager(seed)
81     device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
82
83     # Build env from extracted environment/ files
84     env = _build_env(scenario, seed)
85     if env is None:
86         return {'algo': algo, 'scenario': scenario, 'seed': seed,
87                 'status': 'skipped', 'reason': 'env unavailable'}
88
89     agent = build_rl_agent(algo)
90     result = train_rl_agent(
91         agent, env, scenario, algo, seed,
92         num_episodes=num_ep, device=device,
93         checkpoint_dir=os.path.join(MODELS_DIR, 'checkpoints'))
94
95     result['status'] = 'ok'
96     return result
97
98     except Exception as e:
99         return {'algo': algo, 'scenario': scenario, 'seed': seed,
100                'status': 'error', 'error': str(e)}
101     finally:
102         torch.cuda.empty_cache()
103
104
105 def _diff_worker(job: Dict) -> Dict:
106     """Single diffusion training job - runs in a subprocess."""

```

```

107     import torch
108     os.environ.setdefault('PYTORCH_CUDA_ALLOC_CONF',
109                             'expandable_segments:True')
109     torch.cuda.empty_cache()
110
111     diff_algo = job['diff_algo']
112     rl_algo   = job['rl_algo']
113     scenario  = job['scenario']
114     seed      = job['seed']
115     quick     = job.get('quick', False)
116     num_ep    = 2 if quick else diff_cfg.num_epochs
117
118     try:
119         from utils.reproducibility import ReproducibilityManager
120         from models.rl_agents import build_rl_agent
121         from models.diffusion_models import build_diffusion_model
122         from training.diffusion_trainer import train_diffusion_model
123
124         repro = ReproducibilityManager(seed)
125         device = torch.device('cuda:0' if torch.cuda.is_available() else
126                               'cpu')
127
128         # Load trained RL agent
129         rl_agent = build_rl_agent(rl_algo)
130         ckpt = os.path.join(MODELS_DIR, 'checkpoints',
131                             f'{rl_algo}_{scenario}_seed{seed}_best.pt')
132         if os.path.exists(ckpt):
133             state = torch.load(ckpt, map_location=device)
134             if isinstance(state, dict) and 'model_state' in state:
135                 rl_agent.load_state_dict(state['model_state'])
136             else:
137                 rl_agent.load_state_dict(state)
138         else:
139             return {'status': 'skipped', 'reason': f'RL checkpoint not

```

```

139         found: {ckpt}}'}
140
141         diff_model = build_diffusion_model(diff_algo)
142         env = _build_env(scenario, seed)
143         if env is None:
144             return {'status': 'skipped', 'reason': 'env unavailable'}
145
146         result = train_diffusion_model(
147             diff_model, rl_agent, env, scenario,
148             diff_algo, rl_algo, seed,
149             num_epochs=num_ep, device=device,
150             checkpoint_dir=os.path.join(MODELS_DIR, 'checkpoints'))
151
152         result['status'] = 'ok'
153         return result
154
155     except Exception as e:
156         return {'diff_algo': diff_algo, 'rl_algo': rl_algo, 'scenario':
157             scenario,
158             'seed': seed, 'status': 'error', 'error': str(e)}
159
160     finally:
161         torch.cuda.empty_cache()
162
163
164 def _build_env(scenario: str, seed: int):
165     """
166     Build ImprovedTaskAllocationEnv wrapped with RewardShapingWrapper.
167     Fully self-contained - uses only extracted environment/ files.
168     Falls back to _PlaceholderEnv for smoke testing.
169     """
170     try:
171         from environment.reward_wrapper import build_env
172         return build_env(scenario, seed)
173     except Exception as e:

```

```

171         print(f' Real env unavailable ({e}) --- using placeholder')
172         return _PlaceholderEnv(scenario, seed)
173
174
175
176 class _PlaceholderEnv:
177     """Minimal placeholder environment for import testing."""
178     from config.base_config import (
179         LOCAL_OBS_DIM as _L, FLAT_OBS_DIM as _F,
180         MAX_TASKS as _T, MAX_ROBOTS as _R
181     )
182
183     def __init__(self, scenario, seed):
184         self.scenario = scenario
185         self.num_robots = 5
186         self.tasks = [{'id': i, 'status': 'pending',
187                        'location': np.random.uniform(-10, 10, 2)}
188                        for i in range(10)]
189         self.robots = [{'id': i, 'position': np.random.uniform(-5, 5,
190                        2),
191                        'current_battery': 80.0, 'battery_capacity':
192                        80.0,
193                        'operational_status': ROBOT_STATUS_ACTIVE,
194                        'current_task': None}
195                        for i in range(5)]
196
197     def reset(self, seed=None):
198         self.tasks = [{'id': i, 'status': 'pending',
199                        'location': np.random.uniform(-10, 10, 2)}
200                        for i in range(10)]
201         self.robots = [{'id': i, 'position': np.random.uniform(-5, 5, 2),
202                        'current_battery': 80.0, 'battery_capacity': 80.0,
203                        'operational_status': ROBOT_STATUS_ACTIVE,
204                        'current_task': None}

```

```

201         for i in range(5)]
202         obs = np.random.randn(910).astype(np.float32)
203         return obs, {}
204
205     def step(self, action):
206         obs = np.random.randn(910).astype(np.float32)
207         reward = -1.0
208         done = np.random.random() < 0.01
209         return obs, reward, done, False, {}
210
211 import numpy as np
212
213
214 # Main orchestrator
215
216 def run_all(quick: bool = False, rl_only: bool = False,
217            rl_workers: int = par_cfg.rl_workers,
218            diff_workers: int = par_cfg.diff_workers,
219            seeds: List[int] = None):
220     """
221     Run all 75 RL + 75 Diffusion jobs across 3 seeds.
222
223     Jobs are batched into phases:
224     Phase 1: All RL jobs (75 total: 5 algos × 5 scenarios × 3 seeds)
225     Actually: 5 algos × 3 scenarios × 3 seeds = 45 jobs
226     Phase 2: All Diffusion jobs (5 diff × 5 rl × 3 sc × 3 seeds)
227     Actually: 5 diff × 5 rl × 3 sc × 3 seeds = 225 jobs
228     (We train fewer for speed - see training plan)
229     """
230     seeds = seeds or SEEDS
231     logger = setup_logging()
232     os.makedirs(RESULTS_DIR, exist_ok=True)
233     os.makedirs(MODELS_DIR, exist_ok=True)
234     os.makedirs(SIM_READY_DIR, exist_ok=True)

```



```

235
236     all_results = {}
237     t0 = time.time()
238
239     logger.info('=' * 60)
240     logger.info('AQUA Guardian --- Training Start')
241     logger.info(f'  Scenarios: {SCENARIOS}')
242     logger.info(f'  RL algos:  {RL_ALGOS}')
243     logger.info(f'  Diff algos:{DIFF_ALGOS}')
244     logger.info(f'  Seeds:      {seeds}')
245     logger.info(f'  Quick:      {quick}')
246     logger.info('=' * 60)
247
248     # Phase 1: RL training
249     logger.info('\n=== Phase 1: RL Training ===')
250     rl_jobs = []
251     for algo in RL_ALGOS:
252         for scenario in SCENARIOS:
253             for seed in seeds:
254                 rl_jobs.append({'algo': algo, 'scenario': scenario,
255                                'seed': seed, 'quick': quick})
256
257     logger.info(f'  {len(rl_jobs)} RL jobs, {rl_workers} workers')
258
259     with mp.Pool(processes=rl_workers) as pool:
260         rl_results = pool.map(_rl_worker, rl_jobs)
261
262     for res in rl_results:
263         key =
264         f"RL_{res.get('algo')}-{res.get('scenario')}_s{res.get('seed')}"
265         all_results[key] = res
266         status = res.get('status', '?')
267         if status == 'ok':
268             logger.info(f"  tick {key}  reward={res.get('best_reward',

```

```

0):.1f} "
268             f"completion={res.get('final_completion',
0)*100:.1f}%")
269         else:
270             logger.warning(f"  cross {key}  {res.get('error',
res.get('reason', ''))}")
271
272     # Save Phase 1 checkpoint
273     with open(os.path.join(RESULTS_DIR, 'phase1_rl.pkl'), 'wb') as f:
274         pickle.dump(all_results, f)
275
276     if rl_only:
277         logger.info('--rl_only flag set, stopping after Phase 1')
278         export_results_to_excel(all_results, RESULTS_DIR)
279         return all_results
280
281     # Phase 2: Diffusion training
282     logger.info('\n=== Phase 2: Diffusion Training ===')
283     diff_jobs = []
284     for diff_algo in DIFF_ALGOS:
285         for rl_algo in RL_ALGOS:
286             for scenario in SCENARIOS:
287                 for seed in seeds:
288                     diff_jobs.append({
289                         'diff_algo': diff_algo, 'rl_algo': rl_algo,
290                         'scenario': scenario, 'seed': seed, 'quick': quick
291                     })
292
293     logger.info(f'  {len(diff_jobs)} diffusion jobs, {diff_workers}
workers')
294
295     with mp.Pool(processes=diff_workers) as pool:
296         diff_results = pool.map(_diff_worker, diff_jobs)
297

```

```

298     for res in diff_results:
299         key = (f"DIFF_{res.get('diff_algo')}-{res.get('rl_algo')}_-"
300              f"{res.get('scenario')}_s{res.get('seed')}")
301         all_results[key] = res
302         if res.get('status') == 'ok':
303             logger.info(f"  tick {key}  loss={res.get('best_loss',
304             0):.6f}")
305         else:
306             logger.warning(f"  cross {key}  {res.get('error',
307             res.get('reason', ''))}")
308
309
310     with open(os.path.join(RESULTS_DIR, 'phase2_diffusion.pkl'), 'wb') as
311     f:
312         pickle.dump(all_results, f)
313
314     # Phase 3: Export sim-ready.pt files
315     logger.info('\n=== Phase 3: Exporting Sim-Ready Models ===')
316     _export_all_sim_ready(all_results, logger)
317
318     # Phase 4: Statistical Analysis & Excel Export
319     logger.info('\n=== Phase 4: Statistical Analysis & Excel Export ===')
320     export_results_to_excel(all_results, RESULTS_DIR)
321
322     # Phase 5: Publication figures
323     logger.info('\n=== Phase 5: Generating All 20 Publication Figures
324     ===')
325     try:
326         from training.visualizer import generate_all_figures
327         fig_dir = os.path.join(RESULTS_DIR, 'figures')
328         generated = generate_all_figures(RESULTS_DIR, fig_dir)
329         logger.info(f'  {len(generated)}/20 figures → {fig_dir}')
330     except Exception as ve:
331         logger.warning(f'  Visualisation error: {ve}')
332

```

```

328     total_min = (time.time() - t0) / 60
329     logger.info(f'\nAll done in {total_min:.1f} min')
330     logger.info(f'  Results : {RESULTS_DIR}/training_results.xlsx')
331     logger.info(f'  Figures : {RESULTS_DIR}/figures/')
332     logger.info(f'  Models  : {SIM_READY_DIR}/')
333     return all_results
334
335
336 def _export_all_sim_ready(all_results: Dict, logger) -> None:
337     """Export sim-ready.pt files for all successfully trained RL+Diff
338     combos."""
339     from models.rl_agents import build_rl_agent
340     from models.diffusion_models import build_diffusion_model
341
342     device = torch.device('cpu')    # export on CPU
343
344     for scenario in SCENARIOS:
345         for rl_algo in RL_ALGOS:
346             for diff_algo in DIFF_ALGOS:
347                 # Use seed=42 for the primary sim model
348                 seed = SEEDS[0]
349                 rl_key   = f'RL_{rl_algo}_{scenario}_s{seed}'
350                 diff_key =
351                 f'DIFF_{diff_algo}_{rl_algo}_{scenario}_s{seed}'
352
353                 rl_res   = all_results.get(rl_key, {})
354                 diff_res = all_results.get(diff_key, {})
355
356                 rl_path   = rl_res.get('model_path')
357                 diff_path = diff_res.get('model_path')
358
359                 if not (rl_path and os.path.exists(rl_path) and
360                         diff_path and os.path.exists(diff_path)):
361                     continue

```

```

360
361         try:
362             rl_agent      = build_rl_agent(rl_algo)
363             diff_model    = build_diffusion_model(diff_algo)
364             rl_agent.load_state_dict(torch.load(rl_path,
map_location=device))
365             diff_model.load_state_dict(torch.load(diff_path,
map_location=device))
366
367             rl_sim_path, diff_sim_path = export_sim_ready(
368                 rl_agent, diff_model,
369                 rl_algo, diff_algo, scenario, SIM_READY_DIR)
370
371             logger.info(f'  tick {rl_algo}+{diff_algo}/{scenario}
,
372
f'→ {os.path.basename(rl_sim_path)}')
373         except Exception as e:
374             logger.warning(f'  cross Export failed
{rl_algo}+{diff_algo}/{scenario}: {e}')
375
376
377 if __name__ == '__main__':
378     # Prefer the unified run.py orchestrator - this script is kept for
379     # backwards compatibility with old 'python parallel/job_runner.py
--quick' invocations.
380     parser = argparse.ArgumentParser(description='AQUA Guardian Training
(compat wrapper)')
381     parser.add_argument('--quick',    action='store_true', help='Quick
smoke test')
382     parser.add_argument('--rl_only', action='store_true', help='Train RL
only, skip diffusion')
383     parser.add_argument('--workers', type=int, default=None,
help='Override parallel workers')
384     args = parser.parse_args()

```

```

385
386     # Forward to run.py by calling run.main() directly (avoids sys.argv
        mutation).
387     import warnings
388     warnings.warn(
389         "job_runner.py is deprecated; call 'python run.py' directly.",
390         DeprecationWarning,
391         stacklevel=1,
392     )
393     sys.path.insert(0, str(Path(__file__).parent.parent))
394     import run # noqa: E402
395     # Build a minimal Namespace so run.main() gets the right parameters
        # without touching sys.argv.
396
397     import argparse as _ap
398     _ns = _ap.Namespace(
399         mode='quick' if args.quick else '2hr',
400         rl_only=args.rl_only,
401         workers=args.workers,
402         scenario='all',
403         resume=None,
404     )
405     run.main(_ns)

```

Listing B.31: Multiprocessing job runner that schedules training jobs across worker processes with GPU memory budgeting.

Top-level entry points

```

1  # !/usr/bin/env python3
2  """
3  run.py - AQUA Guardian unified training orchestrator
4  =====
5

```

```

6 Single entry point. No wrapper scripts. Training modes:
7
8 python run.py --mode sanity      # ≤10 min --- verify all code paths
   execute
9 python run.py --mode quick      # ≤10 min --- confirm no errors
   end-to-end
10 python run.py --mode mini       # ≤1 hr  --- confirm models are learning
11 python run.py --mode 2hr        # ~2 h   --- meaningful learning curves
12 python run.py --mode full_train # ~16-20 h --- fully deployable sim model
13 python run.py --mode real_world --rw-scenario warehouse # online robot
   fine-tuning
14
15 orchestration: strict 4-stage dependency pipeline
16 -----
17 Stage A: 4 base RL x 3 scenarios = 12 jobs
18 [ barrier: all of A must succeed ]
19 Stage B: HybridRL x 3 scenarios = 3 jobs
20 (each B job MANDATORILY distills from that scenario's 4 A ckpts)
21 [ barrier: all of B must succeed ]
22 Stage C: 4 base diffusion x HybridRL x 3 scen = 12 jobs
23 (all diffusion trains on top of HybridRL only, per config.DIFF_RL_PAIR)
24 [ per-scenario barrier: all 4 C ckpts for scenario s must succeed ]
25 Stage D: HybridDiffusion x HybridRL x 3 scen = 3 jobs
26 (each D job MANDATORILY distills from that scenario's 4 C ckpts)
27 Total: 30 training jobs
28
29 The previous run.py had two orchestrators (run_sync / run_async). Both
   gone.
30 has exactly one: run_staged.
31
32 defaults baked in (were run_fast.py env vars)
33 -----
34 AQUA_AMP=1 mixed-precision on
35 AQUA_MIN_SNR=1 Min-SNR loss weighting for diffusion

```

```

36  AQUA_NO_AUTO_SHRINK=1 keep chosen batch sizes
37  PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True
38  AQUA_GPU_LOCK_PCT=0.50 reserve half of per-worker VRAM
39  RL batch 512 / Diff batch 256 via mode presets (use --safe for 256 / 128)
40
41  Resume semantics
42  -----
43  Re-running the same command reads stageA_rl.pkl / stageB_hybridrl.pkl /
44  stageC_diff.pkl / stageD_hybriddiff.pkl. Completed jobs are skipped,
    in-flight
45  jobs re-run. Barriers still enforce completion before the next stage
    starts, so
46  a resume can only proceed past a barrier if all earlier-stage jobs are
    'ok'.
47
48  Per-job error isolation
49  -----
50  Each worker process wraps its single training call in one try/except so a
51  failed job returns {'status': 'error',...} and the orchestrator moves on.
52  Every other try/except fallback inside the trainers and models has been
53  removed per the no-fallbacks policy.
54  """
55  from __future__ import annotations
56
57  import argparse
58  import gc
59  import logging
60  import multiprocessing as mp
61  import os
62  import pickle
63  import sys
64  import time
65  import traceback
66  from pathlib import Path

```



```

67 from typing import Any, Dict, List, Optional, Tuple
68
69 import numpy as np
70 import torch
71
72 ROOT = Path(__file__).parent
73 sys.path.insert(0, str(ROOT))
74
75 from config.base_config import ( # noqa: E402
76     DIFF_ALGOS, DIFF_RL_PAIR, LOGS_DIR, MODELS_DIR, MODES, RL_ALGOS,
77     RESULTS_DIR, SCENARIOS, SEEDS, SIM_READY_DIR, apply_mode,
78 )
79
80
81 SEED = SEEDS[0]
82
83 # Module-level logger used by helpers that run inside worker subprocesses.
84 _wlog = logging.getLogger('aqua.worker')
85
86 # Job-count constants (diffusion is HybridRL-only, so 5 diff x 3
    scenarios)
87 BASE_RL_ALGOS = [a for a in RL_ALGOS if a != 'HybridRL'] # 4
88 BASE_DIFF_ALGOS = [d for d in DIFF_ALGOS if d != 'HybridDiffusion'] # 4
89
90 N_STAGE_A = len(BASE_RL_ALGOS) * len(SCENARIOS) # 12
91 N_STAGE_B = len(SCENARIOS) # 3
92 N_STAGE_C = len(BASE_DIFF_ALGOS) * len(SCENARIOS) # 12
93 N_STAGE_D = len(SCENARIOS) # 3
94 N_RL = N_STAGE_A + N_STAGE_B # 15
95 N_DIFF = N_STAGE_C + N_STAGE_D # 15
96
97
98 #
99 # Logging

```

```

100 #
101
102 def setup_logging(mode: str) -> logging.Logger:
103     os.makedirs(LOGS_DIR, exist_ok=True)
104     ts = time.strftime('%Y%m%d_%H%M%S')
105     path = os.path.join(LOGS_DIR, f'run_{mode}_{ts}.log')
106     fmt = '%(asctime)s %(levelname)-8s %(name)s: %(message)s'
107     logging.basicConfig(
108         level=logging.DEBUG,
109         format=fmt,
110         handlers=[
111             logging.FileHandler(path),
112             logging.StreamHandler(),
113         ],
114         force=True,
115     )
116     # Suppress noisy third-party DEBUG logs; keep aqua.* at DEBUG.
117     logging.getLogger('matplotlib').setLevel(logging.WARNING)
118     logging.getLogger('PIL').setLevel(logging.WARNING)
119     return logging.getLogger(f'aqua.{mode}')
120
121
122 def _setup_worker_logging() -> None:
123     """Initialize logging in a spawned worker subprocess.
124
125     Spawn clears all handlers; this adds a stdout StreamHandler so every
126     logger.info/warning call inside a worker shows up with PID context.
127     """
128     root = logging.getLogger()
129     if not root.handlers:
130         h = logging.StreamHandler(sys.stdout)
131         h.setFormatter(logging.Formatter(
132             '%(asctime)s %(levelname)-8s [pid=%(process)d] %(name)s: '
133             '%(message)s'

```

```

133     ))
134     root.addHandler(h)
135     root.setLevel(logging.INFO)
136     # Keep noisy third-party loggers quiet in workers too.
137     logging.getLogger('matplotlib').setLevel(logging.WARNING)
138     logging.getLogger('PIL').setLevel(logging.WARNING)
139
140
141 #
142 # Worker functions (run in spawned subprocesses; must be module-level for
143 spawn)
144 #
145
146 def _setup_worker_env(env_overrides: Dict[str, str]) -> None:
147     """Sync env vars into the child process and reserve its VRAM share."""
148     _setup_worker_logging()
149     for k, v in env_overrides.items():
150         os.environ[k] = v
151     os.environ.setdefault('PYTORCH_CUDA_ALLOC_CONF',
152                          'expandable_segments:True')
153     if torch.cuda.is_available():
154         torch.cuda.empty_cache()
155         from utils.gpu_utils import setup_gpu_memory
156         status = setup_gpu_memory()
157         if status.get('reserved_gb', 0) > 0 and not
158            status.get('already_reserved'):
159             _wlog.info(
160                 f'[worker pid={os.getpid()}] reserved
161                 {status["reserved_gb"]:.2f} GB / '
162                 f'{status.get("cap_gb", 0):.2f} GB cap'
163             )
164
165 def _apply_mode_in_worker(mode: str) -> Dict:

```

```

163     from config.base_config import apply_mode as _apply
164     return _apply(mode)
165
166
167 def rl_worker(job: Dict) -> Dict:
168     """
169     Train one RL agent (base or HybridRL). Runs in a spawned subprocess.
170
171     The wrapping try/except is the per-job error isolation barrier mentioned
172     in the module docstring. It's the ONLY try/except left on the RL training
173     path.
174     """
175     _setup_worker_env(job.get('env_overrides', {}))
176     mode_cfg = _apply_mode_in_worker(job['mode'])
177
178     algo, scenario, seed = job['algo'], job['scenario'], job['seed']
179     num_ep    = job.get('num_episodes', mode_cfg['rl_episodes'])
180     max_st    = job.get('max_steps',    mode_cfg['max_steps'])
181     log_every = mode_cfg.get('log_every', 50)
182
183     try:
184         device = torch.device('cuda:0' if torch.cuda.is_available() else
185                               'cpu')
186
187         from models.rl_agents import build_rl_agent
188         from training.rl_trainer import train_rl_agent
189         from environment.reward_wrapper import build_env
190         from utils.reproducibility import ReproducibilityManager
191         from config.base_config import RLConfig as _RC
192
193         ReproducibilityManager(seed)
194
195         _RC.max_steps = max_st
196         _RC.eps_decay = max(100, int(_RC.eps_decay * num_ep / 5000))

```

```

196
197     env    = build_env(scenario, seed)
198     agent = build_rl_agent(algo)
199
200     # Stage-B distillation: HybridRL MUST warm-start from the 4
base-algo
201     # checkpoints trained in Stage A for this scenario. No longer
optional.
202     if algo == 'HybridRL':
203         _distill_hybrid_rl(agent, scenario, seed, device)
204
205     ckpt_dir = os.path.join(MODELS_DIR, 'checkpoints')
206     os.makedirs(ckpt_dir, exist_ok=True)
207
208     # SAR: route to scenario-specific trainer based on agent_mode
config flag
209     from config.base_config import SCENARIO_CONFIGS as _SC
210     if scenario == 'search_and_rescue':
211         _agent_mode = _SC['search_and_rescue'].agent_mode
212         if _agent_mode == 'drone':
213             from training.sar_drone_trainer import train_sar_drone
214             result = train_sar_drone(
215                 agent, env, seed, algo=algo,
216                 num_episodes=num_ep, device=device,
217                 checkpoint_dir=ckpt_dir, log_every=log_every,
218             )
219         else:
220             from training.sar_land_robot_trainer import
train_sar_land_robot
221             result = train_sar_land_robot(
222                 agent, env, seed, algo=algo,
223                 num_episodes=num_ep, device=device,
224                 checkpoint_dir=ckpt_dir, log_every=log_every,
225             )

```

```

226         else:
227             result = train_rl_agent(
228                 agent, env, scenario, algo, seed,
229                 num_episodes=num_ep, device=device,
230                 checkpoint_dir=ckpt_dir, log_every=log_every,
231             )
232             result.setdefault('algo',      algo)
233             result.setdefault('scenario',  scenario)
234             result.setdefault('seed',     seed)
235             result.setdefault('status',   'ok')
236             return result
237
238     except Exception as e:
239         return {
240             'algo': algo, 'scenario': scenario, 'seed': seed,
241             'status': 'error',
242             'error': f'{e}\n{traceback.format_exc()[:1200]}',
243         }
244     finally:
245         if torch.cuda.is_available():
246             torch.cuda.empty_cache()
247         gc.collect()
248
249
250 def diff_worker(job: Dict) -> Dict:
251     """
252     Train one diffusion model (base or HybridDiffusion).
253
254     The RL checkpoint to condition on is ALWAYS DIFF_RL_PAIR (i.e. HybridRL).
255     Stage C and D barriers guarantee the ckpt exists, so the previous
256     ckpt-missing try/except that returned status='skipped' has been removed.
257     If the ckpt is missing now, that's a pipeline invariant violation and the
258     job errors.
259     """

```

```

260     _setup_worker_env(job.get('env_overrides', {}))
261     mode_cfg = _apply_mode_in_worker(job['mode'])
262
263     da, ra, sc, seed = job['diff_algo'], job['rl_algo'], job['scenario'],
264         job['seed']
265     num_ep = job.get('num_epochs', mode_cfg['diff_epochs'])
266     n_coll = job.get('num_collect', mode_cfg['diff_collect'])
267
268     try:
269         device = torch.device('cuda:0' if torch.cuda.is_available() else
270             'cpu')
271
272         from models.rl_agents import build_rl_agent
273         from models.diffusion_models import build_diffusion_model
274         import training.diffusion_trainer as dt
275         from environment.reward_wrapper import build_env
276         from utils.reproducibility import ReproducibilityManager
277         from config.base_config import DiffusionConfig as _DC
278
279         ReproducibilityManager(seed)
280         _DC.traj_collect_episodes = int(n_coll)
281
282         # Load the HybridRL checkpoint that Stage B produced. Must exist.
283         rl_agent = build_rl_agent(ra)
284         ckpt_dir = os.path.join(MODELS_DIR, 'checkpoints')
285         rl_ckpt_path = _find_rl_ckpt(ckpt_dir, ra, sc, seed)
286         if rl_ckpt_path is None:
287             raise FileNotFoundError(
288                 f'Stage-B invariant violated: no RL checkpoint at '
289                 f'{ckpt_dir}/{ra}_{sc}_seed{seed}_{{"best", "final"}}.pt')
290         state = torch.load(rl_ckpt_path, map_location=device,
291             weights_only=False)
292         sd = state.get('model_state', state.get('model_state_dict',
293             state))\

```

```

290         if isinstance(state, dict) else state
291         rl_agent.load_state_dict(sd, strict=False) # allow checkpoint
keys absent from fresh model
292
293         diff_model = build_diffusion_model(da)
294
295         # Stage-D distillation: HybridDiffusion MUST warm-start from the 4
# base-diffusion checkpoints trained in Stage C for this scenario.
296         if da == 'HybridDiffusion':
297             _distill_hybrid_diff(diff_model, ra, sc, seed, device)
298
299
300         env = build_env(sc, seed)
301         result = dt.train_diffusion_model(
302             diff_model, rl_agent, env, sc, da, ra, seed,
303             num_epochs=num_ep, device=device, checkpoint_dir=ckpt_dir,
304         )
305         result.setdefault('diff_algo', da)
306         result.setdefault('rl_algo', ra)
307         result.setdefault('scenario', sc)
308         result.setdefault('seed', seed)
309         result.setdefault('status', 'ok')
310         return result
311
312     except Exception as e:
313         return {
314             'diff_algo': da, 'rl_algo': ra, 'scenario': sc, 'seed': seed,
315             'status': 'error',
316             'error': f'{e}\n{traceback.format_exc()[:1200]}' ,
317         }
318     finally:
319         if torch.cuda.is_available():
320             torch.cuda.empty_cache()
321         gc.collect()
322

```



```

323
324 #
325 # Checkpoint helpers
326 #
327
328 def _find_rl_ckpt(ckpt_dir: str, algo: str, scenario: str, seed: int) ->
    Optional[str]:
329     """Return path to an existing RL ckpt (_best.pt preferred) or None."""
330     for suf in ('_best.pt', '_final.pt'):
331         p = os.path.join(ckpt_dir, f'{algo}_{scenario}_seed{seed}{suf}')
332         if os.path.exists(p):
333             return p
334     return None
335
336
337 def _find_diff_ckpt(ckpt_dir: str, diff: str, rl: str,
338                     scenario: str, seed: int) -> Optional[str]:
339     """Return path to an existing diffusion ckpt (_best.pt preferred) or
    None."""
340     for suf in ('_best.pt', '_final.pt'):
341         p = os.path.join(ckpt_dir,
342                         f'{diff}_{rl}_{scenario}_seed{seed}{suf}')
343         if os.path.exists(p):
344             return p
345     return None
346
347 #
348 # Mandatory hybrid distillation
349 #
350
351 def _distill_hybrid_rl(hybrid_agent, scenario: str, seed: int, device) ->
    None:
352     """

```

```

353 Warm-start HybridRL sub-modules from the 4 base-algo checkpoints trained
354 in Stage A for this scenario. This is MANDATORY (Stage B only runs
355 after Stage A barrier, so all 4 ckpts are guaranteed to exist). Previous
356 silent try/except per ckpt is removed.
357 """
358     ckpt_dir = os.path.join(MODELS_DIR, 'checkpoints')
359     sub_map = {
360         'MAPPO': ['mappo_actor', 'mappo_critic'],
361         'QMIX': ['qmix_agent'],
362         'MADDPG': ['maddpg_actor'],
363         'DQN': ['dqn_encoder', 'dqn_head'],
364     }
365     target_sd = hybrid_agent.state_dict()
366     n_loaded = 0
367     for base_algo, prefixes in sub_map.items():
368         path = _find_rl_ckpt(ckpt_dir, base_algo, scenario, seed)
369         if path is None:
370             raise FileNotFoundError(
371                 f'Stage-A invariant violated: no {base_algo} ckpt for
372                 {scenario}. '
373                 f'HybridRL distillation requires all 4 base-algo Stage-A
374                 ckpts.')
375         state = torch.load(path, map_location=device, weights_only=False)
376         sd = state.get('model_state', state.get('model_state_dict',
377             state))\
378             if isinstance(state, dict) else state
379         # the base-agent keys. MAPPO's 'actor.actor.encoder.net.0.weight'
380         # corresponds to HybridRL's
381         'mappo_actor.actor.encoder.net.0.weight'
382         # (single 'actor' under the prefix, not double). Strip the first
383         # dotted component of each base-agent key before prefixing.
384         # SENTINEL: distill-strip-v5
385         for k, v in sd.items():
386             stripped = k.split('.', 1)[1] if '.' in k else k

```

```

383         for pfx in prefixes:
384             tgt_key = f'{pfx}.{stripped}'
385             if tgt_key in target_sd and target_sd[tgt_key].shape ==
v.shape:
386                 target_sd[tgt_key] = v
387                 n_loaded += 1
388                 break
389     hybrid_agent.load_state_dict(target_sd, strict=False)
390     _wlog.info(f'[distill] HybridRL({scenario}): warm-started {n_loaded}
tensors '
391             f'from 4 base-algo checkpoints')
392
393
394 def _distill_hybrid_diff(hybrid_diff, rl_algo: str, scenario: str,
395                          seed: int, device) -> None:
396     """
397     Warm-start HybridDiffusion sub-modules from the 4 base-diff checkpoints
398     trained in Stage C for this scenario. MANDATORY.
399     """
400     ckpt_dir = os.path.join(MODELS_DIR, 'checkpoints')
401     sub_names = {
402         'TDM': 'tdm',
403         'DiffusionPolicy': 'dp',
404         'Diffuser': 'diff',
405         'SDE': 'sde',
406     }
407     target_sd = hybrid_diff.state_dict()
408     n_loaded = 0
409     for base_diff, sub_prefix in sub_names.items():
410         path = _find_diff_ckpt(ckpt_dir, base_diff, rl_algo, scenario,
seed)
411         if path is None:
412             raise FileNotFoundError(
413                 f'Stage-C invariant violated: no {base_diff} ckpt for '

```

```

414         f'{rl_algo}/{scenario}. HybridDiffusion distillation
requires '
415         f'all 4 base-diffusion Stage-C ckpts for the scenario.')
```

sd = torch.load(path, map_location=device, weights_only=False)

if isinstance(sd, dict) and 'model_state' in sd:

sd = sd['model_state']

for k, v in sd.items():

tgt_key = f'{sub_prefix}.{k}'

if tgt_key in target_sd and target_sd[tgt_key].shape ==

v.shape:

target_sd[tgt_key] = v

n_loaded += 1

hybrid_diff.load_state_dict(target_sd, strict=False)

_wlog.info(f'[distill] HybridDiff({rl_algo}/{scenario}): warm-started

,

f'{n_loaded} tensors from 4 base-diff checkpoints')

#

Env-var defaults (baked-in from run_fast.py)

#

def _build_env_overrides(args) -> Dict[str, str]:

"""

Construct the env-var dict that every worker inherits.

These are the run_fast.py defaults, now first-class in run.py.

"""

ov: Dict[str, str] = {

'PYTORCH_CUDA_ALLOC_CONF': 'expandable_segments:True',

'AQUA_NO_AUTO_SHRINK': '1',

OOM-retry removed in. Kept env var at 0

for any dependent code that checks it.

'AQUA_OOM_RETRY': '0',

}

```

445     ov['AQUA_AMP']      = '0' if args.no_amp else '1'
446     ov['AQUA_MIN_SNR'] = '0' if args.no_amp else '1'    # tied to AMP
447     if args.gpu_lock_pct is not None and args.gpu_lock_pct > 0:
448         ov['AQUA_GPU_LOCK_PCT'] = f'{args.gpu_lock_pct:.3f}'
449     if getattr(args, 'per_worker_gb', None):
450         ov['AQUA_GPU_GB_PER_WORKER'] = f'{args.per_worker_gb:.3f}'
451     if args.rl_batch:
452         ov['AQUA_RL_BATCH_SIZE'] = str(args.rl_batch)
453     if args.diff_batch:
454         ov['AQUA_DIFF_BATCH_SIZE'] = str(args.diff_batch)
455     return ov
456
457
458 #
459 # Stage orchestration
460 #
461
462 _STAGE_PKL = {
463     'A': 'stageA_rl.pkl',
464     'B': 'stageB_hybridrl.pkl',
465     'C': 'stageC_diff.pkl',
466     'D': 'stageD_hybriddiff.pkl',
467 }
468
469
470 def _save_stage(results: Dict, prefix_filter, filename: str) -> None:
471     subset = {k: v for k, v in results.items() if prefix_filter(k)}
472     with open(os.path.join(RESULTS_DIR, filename), 'wb') as f:
473         pickle.dump(subset, f)
474
475
476 def _load_stage(filename: str) -> Dict:
477     path = os.path.join(RESULTS_DIR, filename)
478     if not os.path.exists(path):

```

```

479         return {}
480     with open(path, 'rb') as f:
481         return pickle.load(f)
482
483
484 def _handle_rl_result(r: Dict, results: Dict, logger) -> None:
485     a = r.get('algo') or '?'
486     sc = r.get('scenario') or '?'
487     sd = r.get('seed', SEED)
488     key = f"RL_{a}_{sc}_s{sd}"
489     results[key] = r
490     status = r.get('status', '?')
491     if status == 'ok':
492         compl = r.get('final_completion', 0) or 0
493         rew = r.get('best_reward', 0)
494         rew_str = (f'{rew:8.2f}' if isinstance(rew, (int, float))
495                   and rew != float('-inf') else '    n/a')
496         logger.info(f"   RL   {a:<10s} {sc:<20s} "
497                   f"compl={compl*100:5.1f}% reward={rew_str}")
498     else:
499         logger.warning(f" - RL   {a} {sc}: "
500                   f"{str(r.get('error', r.get('reason',
501                   '')))[:200]}")
501
502
503 def _handle_diff_result(r: Dict, results: Dict, logger) -> None:
504     da = r.get('diff_algo') or '?'
505     ra = r.get('rl_algo') or '?'
506     sc = r.get('scenario') or '?'
507     sd = r.get('seed', SEED)
508     key = f"DIFF_{da}_{ra}_{sc}_s{sd}"
509     results[key] = r
510     status = r.get('status', '?')
511     if status == 'ok':

```

```

512         loss = r.get('best_loss')
513         loss_str = (f'{loss:.6f}' if isinstance(loss, (int, float))
514                     and loss != float('inf') else 'n/a')
515         logger.info(f" + DIFF {da:<16s} {ra:<10s} {sc:<20s}
loss={loss_str}")
516     else:
517         logger.warning(f" - DIFF {da}/{ra}/{sc}: "
518                        f"{str(r.get('error', r.get('reason',
519                        '')))[:200]}")
519
520
521 def _run_pool(jobs: List[Dict], worker_fn, workers: int,
522              handle_result, results: Dict, logger) -> None:
523     """Run a list of jobs in a spawn pool, handling each result as it
524     arrives.
525     When workers==1 jobs are run in-process to avoid subprocess memory
526     overhead."""
527     if not jobs:
528         return
529     if workers == 1:
530         for job in jobs:
531             r = worker_fn(job)
532             handle_result(r, results, logger)
533         return
534     with mp.Pool(workers) as pool:
535         for r in pool.imap_unordered(worker_fn, jobs):
536             handle_result(r, results, logger)
537
538 def _check_barrier(results: Dict, expected_keys: List[str],
539                  stage_name: str, logger) -> None:
540     """
541     Verify every expected key has status=='ok'. Raises RuntimeError otherwise

```

```

541 (hard failure - we don't silently run the next stage on a broken
    pipeline).
542 """
543     missing = [k for k in expected_keys if results.get(k,
    {}).get('status') != 'ok']
544     if missing:
545         logger.error(f'\n *** BARRIER FAILURE: end of {stage_name} ***')
546         for k in missing:
547             r = results.get(k, {})
548             logger.error(f'    {k}: status={r.get("status", "missing")} '
549                         f'err={str(r.get("error", ""))[:180]}')
550         raise RuntimeError(
551             f'Stage {stage_name} barrier:
552             {len(missing)}/{len(expected_keys)} '
553             f'jobs not ok. See log above. Cannot proceed.')
554         logger.info(f' [barrier] {stage_name}: all {len(expected_keys)} jobs
555         ok, '
556
557             f'proceeding.')
```

```

557 def run_staged(mode: str, mode_cfg: Dict, args, logger) -> Dict:
558     """The one and only orchestrator. Four stages, three barriers."""
559     results: Dict[str, Any] = {}
560     results.update(_load_stage(_STAGE_PKL['A']))
561     results.update(_load_stage(_STAGE_PKL['B']))
562     results.update(_load_stage(_STAGE_PKL['C']))
563     results.update(_load_stage(_STAGE_PKL['D']))
564
565     env_ov = _build_env_overrides(args)
566     workers = args.workers or mode_cfg['num_workers']
567     active_sc = getattr(args, 'active_scenarios', SCENARIOS)
568
569     def _mk_rl_job(algo: str, sc: str) -> Dict:
570         return {
```



```

571         'algo': algo, 'scenario': sc, 'seed': SEED,
572         'num_episodes': args.rl_episodes or mode_cfg['rl_episodes'],
573         'max_steps': mode_cfg['max_steps'],
574         'mode': mode, 'env_overrides': env_ov,
575     }
576
577 def _mk_diff_job(da: str, ra: str, sc: str) -> Dict:
578     return {
579         'diff_algo': da, 'rl_algo': ra, 'scenario': sc, 'seed': SEED,
580         'num_epochs': args.diff_epochs or mode_cfg['diff_epochs'],
581         'num_collect': mode_cfg['diff_collect'],
582         'mode': mode, 'env_overrides': env_ov,
583     }
584
585     # Stage A: 4 base RL algos x active scenarios
586     n_a = len(BASE_RL_ALGOS) * len(active_sc)
587     logger.info(f'\n>>> STAGE A : base RL ({n_a} jobs,
588               workers={workers})')
589     t_a = time.time()
590     a_keys = [f'RL_{a}_{s}_{s}{SEED}' for a in BASE_RL_ALGOS for s in
591               active_sc]
592     a_jobs = [_mk_rl_job(a, s) for a in BASE_RL_ALGOS for s in active_sc
593               if results.get(f'RL_{a}_{s}_{s}{SEED}', {}).get('status') !=
594               'ok']
595     logger.info(f' {len(a_jobs)} new, {N_STAGE_A - len(a_jobs)} resumed')
596     _run_pool(a_jobs, rl_worker, workers, _handle_rl_result, results,
597               logger)
598     _save_stage(results, lambda k: k.startswith('RL_') and
599               not k.startswith('RL_HybridRL_'), _STAGE_PKL['A'])
600     logger.info(f' Stage A: {(time.time()-t_a)/60:.1f} min')
601     _check_barrier(results, a_keys, 'A', logger)
602
603     if args.rl_only and args.stop_after == 'A':
604         summarize(results, logger, mode)

```

```

601         return results
602
603     # Stage B: HybridRL x 3 scenarios (mandatory distillation from A)
604     logger.info(f'\n>>> STAGE B : HybridRL ({N_STAGE_B} jobs,
605         workers={workers})')
606     t_b = time.time()
607     b_keys = [f'RL_HybridRL_{s}_{SEED}' for s in active_sc]
608     b_jobs = [_mk_rl_job('HybridRL', s) for s in active_sc
609         if results.get(f'RL_HybridRL_{s}_{SEED}',
610             {}).get('status') != 'ok']
611     logger.info(f' {len(b_jobs)} new, {len(active_sc) - len(b_jobs)}
612         resumed')
613     _run_pool(b_jobs, rl_worker, workers, _handle_rl_result, results,
614         logger)
615     _save_stage(results, lambda k: k.startswith('RL_HybridRL_'),
616         _STAGE_PKL['B'])
617     logger.info(f' Stage B: {(time.time()-t_b)/60:.1f} min')
618     _check_barrier(results, b_keys, 'B', logger)
619
620     if args.rl_only:
621         summarize(results, logger, mode)
622         return results
623
624     # Stage C: 4 base diff x HybridRL x active scenarios
625     n_c = len(BASE_DIFF_ALGOS) * len(active_sc)
626     logger.info(f'\n>>> STAGE C : base diffusion on {DIFF_RL_PAIR} ',
627         f'({n_c} jobs, workers={workers})')
628     t_c = time.time()
629     c_keys = [f'DIFF_{da}_{DIFF_RL_PAIR}_{s}_{SEED}'
630         for da in BASE_DIFF_ALGOS for s in active_sc]
631     c_jobs = [_mk_diff_job(da, DIFF_RL_PAIR, s)
632         for da in BASE_DIFF_ALGOS for s in active_sc
633         if results.get(f'DIFF_{da}_{DIFF_RL_PAIR}_{s}_{SEED}',
634             {}).get('status') != 'ok']

```

```

630     logger.info(f'  {len(c_jobs)} new, {n_c - len(c_jobs)} resumed')
631     _run_pool(c_jobs, diff_worker, workers, _handle_diff_result, results,
        logger)
632     _save_stage(results, lambda k: k.startswith('DIFF_') and
633                 not k.startswith('DIFF_HybridDiffusion_'),
        _STAGE_PKL['C'])
634     logger.info(f'  Stage C: {(time.time()-t_c)/60:.1f} min')
635     _check_barrier(results, c_keys, 'C', logger)
636
637     # Stage D: HybridDiffusion x HybridRL x active scenarios
638     n_d = len(active_sc)
639     logger.info(f'\n>>> STAGE D : HybridDiffusion on {DIFF_RL_PAIR} '
640               f'({n_d} jobs, workers={workers})')
641     t_d = time.time()
642     d_keys = [f'DIFF_HybridDiffusion_{DIFF_RL_PAIR}_{s}_s{SEED}' for s in
        active_sc]
643     d_jobs = [_mk_diff_job('HybridDiffusion', DIFF_RL_PAIR, s) for s in
        active_sc]
644     if
645         results.get(f'DIFF_HybridDiffusion_{DIFF_RL_PAIR}_{s}_s{SEED}',
        {}).get('status') != 'ok']
646     logger.info(f'  {len(d_jobs)} new, {n_d - len(d_jobs)} resumed')
647     _run_pool(d_jobs, diff_worker, workers, _handle_diff_result, results,
        logger)
648     _save_stage(results, lambda k: k.startswith('DIFF_HybridDiffusion_'),
649                 _STAGE_PKL['D'])
650     logger.info(f'  Stage D: {(time.time()-t_d)/60:.1f} min')
651     _check_barrier(results, d_keys, 'D', logger)
652
653     return results
654
655
656 #
657 # Export + summary

```

```

658 #
659
660 def export_all(logger) -> int:
661     """
662     Export sim-ready TorchScript.pt files. only DIFF_RL_PAIR (HybridRL)
663     is paired with diffusion, so we export:
664     - 5 RL.pt (one per algo per scenario = 15)
665     - 5 diff.pt (one per diff algo per scenario = 15)
666     Total 30 sim-ready files.
667     """
668     from models.rl_agents import build_rl_agent
669     from models.diffusion_models import build_diffusion_model
670     from models.sim_ready_wrappers import export_sim_ready
671
672     dev = torch.device('cpu')
673     ckpt_dir = os.path.join(MODELS_DIR, 'checkpoints')
674     n = 0
675     # RL exports: every algo x every scenario
676     for sc in SCENARIOS:
677         for ra in RL_ALGOS:
678             rl_ckpt = _find_rl_ckpt(ckpt_dir, ra, sc, SEED)
679             if rl_ckpt is None:
680                 logger.debug(f' export skip RL {ra}/{sc}: no ckpt')
681                 continue
682             rl = build_rl_agent(ra)
683             st = torch.load(rl_ckpt, map_location=dev, weights_only=False)
684             rl.load_state_dict(st.get('model_state', st)
685                                   if isinstance(st, dict) else st)
686             # Pass a null diff so export_sim_ready only writes the RL file
687             export_sim_ready(rl, None, ra, None, sc, SIM_READY_DIR)
688             n += 1
689     # Diff exports: every diff algo x HybridRL x every scenario
690     for sc in SCENARIOS:
691         for da in DIFF_ALGOS:

```

```

692         diff_ckpt = _find_diff_ckpt(ckpt_dir, da, DIFF_RL_PAIR, sc,
SEED)
693         if diff_ckpt is None:
694             logger.debug(f' export skip DIFF
{da}/{DIFF_RL_PAIR}/{sc}: no ckpt')
695             continue
696             dm = build_diffusion_model(da)
697             st = torch.load(diff_ckpt, map_location=dev,
weights_only=False)
698             dm.load_state_dict(st.get('model_state', st)
699                                 if isinstance(st, dict) else st)
700             export_sim_ready(None, dm, None, da, sc, SIM_READY_DIR,
701                             paired_rl=DIFF_RL_PAIR)
702             n += 1
703
704     logger.info(f' Exported {n} sim-ready .pt files '
705               f'(expected up to {len(SCENARIOS) * (len(RL_ALGOS) +
706               len(DIFF_ALGOS))})')
707
708     return n
709
710 def summarize(all_results: Dict, logger, mode: str) -> None:
711     """
712     Print the final results table. layout reflects the 15-diff grid:
713     rows by algo (RL) or diff-algo, columns by scenario, diffusion is all
714     conditioned on HybridRL.
715     """
716     logger.info('\n' + '=' * 90)
717     logger.info(f'RESULTS - mode={mode} seed={SEED}')
718     logger.info(f'          - 4-stage orchestrator | '
719               f'{N_STAGE_A}+{N_STAGE_B}+{N_STAGE_C}+{N_STAGE_D} = 30
720               jobs')
721     logger.info('=' * 90)

```

```

721     # RL: final completion rates
722     logger.info(f'\n RL FINAL COMPLETION (rows: algo, cols: scenario)')
723     logger.info(f'  {"Algo":<12s} {"Warehouse":>12s} {"SAR":>12s}
{"Maritime":>12s}')
724     logger.info('  ' + '-' * 52)
725     for algo in RL_ALGOS:
726         row = f'  {algo:<12s}'
727         for sc in SCENARIOS:
728             r = all_results.get(f'RL_{algo}_{sc}_s{SEED}', {})
729             if r.get('status') == 'ok':
730                 c = (r.get('final_completion', 0) or 0) * 100
731                 flag = '+' if c >= 50 else '~' if c >= 20 else '!' if c
>= 5 else '.'
732                 row += f'  {flag} {c:5.1f}%  '
733             else:
734                 row += f'  x N/A  '
735             logger.info(row)
736
737     # RL: early-vs-late learning trend
738     logger.info(f'\n RL LEARNING TREND (early quintile -> late
quintile)')
739     logger.info(f'  {"Algo":<12s} {"Scenario":<20s}  '
f'{"Early%":>7s} {"Late%":>7s} {"DCompl":>8s}  '
f'{"Rew_E":>9s} {"Rew_L":>9s} {"DRew":>9s}
{"Learning?"}')
742     logger.info('  ' + '-' * 100)
743     learning_count = 0
744     for algo in RL_ALGOS:
745         for sc in SCENARIOS:
746             r = all_results.get(f'RL_{algo}_{sc}_s{SEED}', {})
747             if r.get('status') != 'ok':
748                 logger.info(f'  {algo:<12s} {sc:<20s} (no result)')
749                 continue
750             comps = r.get('completion_rates', []) or []

```

```

751         rews = r.get('rewards',          []) or []
752         if len(comps) < 5 or len(rews) < 5:
753             logger.info(f' {algo:<12s} {sc:<20s} (too few episodes
to judge)')
754             continue
755         n = max(1, len(comps) // 5)
756         ec = float(np.mean(comps[:n])) * 100
757         lc = float(np.mean(comps[-n:])) * 100
758         er = float(np.mean(rews[:n]))
759         lr = float(np.mean(rews[-n:]))
760         dC = lc - ec
761         dR = lr - er
762         is_learning = (dC > 1.0) or (dR > 1.0)
763         if is_learning:
764             learning_count += 1
765         logger.info(
766             f' {algo:<12s} {sc:<20s} '
767             f'{ec:6.1f}% {lc:6.1f}% {dC:+7.1f}pp '
768             f'{er:9.2f} {lr:9.2f} {dR:+9.2f} '
769             f'{"YES" if is_learning else "no"}')
770
771         # Diffusion: best loss per diff algo per scenario (all on HybridRL)
772         logger.info(f'\n DIFFUSION BEST LOSS (all conditioned on
{DIFF_RL_PAIR})')
773         logger.info(f' {"Diff algo":<18s} {"Warehouse":>12s} {"SAR":>12s} '
774             f'{"Maritime":>12s}')
775         logger.info(' ' + '-' * 60)
776         ok_dd = 0
777         for da in DIFF_ALGOS:
778             row = f' {da:<18s}'
779             for sc in SCENARIOS:
780                 r = all_results.get(f'DIFF_{da}_{DIFF_RL_PAIR}_{sc}_s{SEED}',
{}))
781                 if r.get('status') == 'ok':

```

```

782         ok_dd += 1
783         loss = r.get('best_loss')
784         if isinstance(loss, (int, float)) and loss !=
float('inf'):
785             row += f'   {loss:10.4f} '
786         else:
787             row += f'   {"n/a":>10s} '
788         else:
789             row += f'   {"x":>10s} '
790         logger.info(row)
791
792     # Totals
793     ok_rl = sum(1 for k, v in all_results.items()
794                 if k.startswith('RL_') and v.get('status') == 'ok')
795     pt = (list(Path(SIM_READY_DIR).glob('*.*pt'))
796           if os.path.exists(SIM_READY_DIR) else [])
797
798     logger.info(f'\n  TOTALS')
799     logger.info(f'    RL ok           : {ok_rl} / {N_RL}')
800     logger.info(f'    Diff ok          : {ok_dd} / {N_DIFF}')
801     logger.info(f'    RL learning      : {learning_count} / {N_RL} '
802                 f'(needs DComp1>1pp or DRew>1 for "YES"')
803     logger.info(f'    Sim .pt files   : {len(pt)}')
804     logger.info('=' * 90)
805
806
807 #
808 # Real-world online training
809 #
810
811 def _run_real_world(args, logger) -> None:
812     """Entry point for real-world online RL.
813

```



```

814     Loads a pretrained model (or starts fresh), wraps a RealWorldEnv
        around
815     the user-supplied sensor/command callbacks, and runs the training
        loop.

816
817     The offline pipeline (run_staged) is never called from this path.
818
819     Sensor callbacks:  the stubs below return empty lists so the command
        runs
820     without hardware.  Integrators replace them with real sensor reads
821     (ROS2 subscribers, serial drivers, REST clients, etc.).
822     """
823     from models.rl_agents import build_rl_agent
824     from environment.real_world_env import RealWorldEnv
825     from utils.safety_layer import SafetyLayer
826     from utils.real_world_collector import OnlineTrainer
827
828     # Resolve scenario alias
829     scenario = _SCENARIO_MAP.get(args.rw_scenario or '', args.rw_scenario)
830     if not scenario:
831         logger.error('--rw-scenario is required with --real-world')
832         return
833
834     algo = args.rw_algo or 'HybridRL'
835     device = torch.device('cuda:0' if torch.cuda.is_available() else
        'cpu')
836
837     logger.info('=' * 70)
838     logger.info(f'AQUA Guardian | REAL-WORLD online training')
839     logger.info(f'  scenario   : {scenario}')
840     logger.info(f'  algorithm  : {algo}')
841     logger.info(f'  device    : {device}')
842     logger.info(f'  checkpoint: {args.rw_checkpoint or "none (fresh
        weights)"}')

```

```

843     logger.info(f'  episodes   : {args.rw_steps}')
844     logger.info(f'  step_dt    : {args.rw_step_dt} s')
845     logger.info(f'  async      : {args.rw_async}')
846     logger.info('=' * 70)
847
848     # -----
849     # Sensor / command stubs --- REPLACE these with real hardware drivers
850     # -----
851
852     def robot_state_fn():
853         """Return current robot state list from real sensors."""
854         # Example ROS2 integration:
855         #   return ros_bridge.get_robot_states()
856         return []
857
858     def task_state_fn():
859         """Return current task state list from real sensors."""
860         # Example:
861         #   return task_manager.get_pending_tasks()
862         return []
863
864     def command_fn(robot_id: int, task_id: int) -> None:
865         """Send assignment command to physical robot."""
866         # Example ROS2 action call:
867         #   ros_bridge.assign_task(robot_id, task_id)
868         logger.debug(f'[command] assign robot {robot_id} -> task
869         {task_id}')
870
871     # -----
872     # Build env, agent, safety layer
873     # -----
874
875     safety = SafetyLayer()
876     env     = RealWorldEnv(

```

```

876         scenario          = scenario,
877         robot_state_fn     = robot_state_fn,
878         task_state_fn      = task_state_fn,
879         command_fn         = command_fn,
880         step_dt            = args.rw_step_dt,
881         max_steps          = getattr(args, 'rw_steps', 500),
882         safety_layer       = safety,
883     )
884
885     agent = build_rl_agent(algo)
886
887     ckpt_dir = os.path.join(MODELS_DIR, 'checkpoints', 'realworld')
888     os.makedirs(ckpt_dir, exist_ok=True)
889
890     def _on_episode_end(ep: int, reward: float, comp: float) -> None:
891         if ep % max(1, args.rw_save_every) == 0 and ep > 0:
892             p = os.path.join(ckpt_dir, f'{algo}_{scenario}_ep{ep}.pt')
893             torch.save({'model_state': agent.state_dict()}, p)
894             logger.info(
895                 f'[rw] ep={ep:4d}  reward={reward:8.2f}
896                 comp={comp*100:5.1f}%  '
897                 f'ckpt saved')
898         else:
899             logger.info(
900                 f'[rw] ep={ep:4d}  reward={reward:8.2f}
901                 comp={comp*100:5.1f}%')
902
903     trainer = OnlineTrainer(
904         agent          = agent,
905         env            = env,
906         scenario       = scenario,
907         algo           = algo,
908         device         = device,
909         checkpoint_path = args.rw_checkpoint,

```

```

908         update_every      = args.rw_update_every,
909         use_async          = args.rw_async,
910         safety_layer       = safety,
911         on_episode_end     = _on_episode_end,
912     )
913
914     # Run preflight safety check
915     try:
916         obs, _ = env.reset()
917         issues = safety.validate_fleet(env.robots)
918         if issues:
919             logger.warning(f'[SafetyLayer] preflight issues: {issues}')
920         else:
921             logger.info('[SafetyLayer] preflight OK')
922     except Exception as e:
923         logger.error(f'[rw] preflight failed: {e}')
924         return
925
926     # Execute training
927     if args.rw_async:
928         trainer.run_async(args.rw_steps, checkpoint_dir=ckpt_dir)
929         logger.info('[rw] async threads launched --- running in
background')
930     else:
931         trainer.run_sync(args.rw_steps, checkpoint_dir=ckpt_dir)
932         logger.info('[rw] sync training complete')
933
934
935     #
936     # Main
937     #
938
939     _MODE_ALIASES: Dict[str, str] = {
940         'smoke_test': 'quick',

```

```

941     'mini_train': 'mini',
942     # 'full_train' is now a proper MODES entry; no alias needed.
943     # 'full' is kept in MODES as a backward-compat alias for full_train.
944 }
945 _SCENARIO_MAP: Dict[str, str] = {
946     'sar': 'search_and_rescue',
947     'maritime': 'maritime_security',
948     'warehouse': 'warehouse',
949     'search_and_rescue': 'search_and_rescue',
950     'maritime_security': 'maritime_security',
951 }
952
953
954 def main(args=None):
955     _all_modes = list(MODES.keys()) + list(_MODE_ALIASES.keys())
956     parser = argparse.ArgumentParser(
957         description='AQUA Guardian unified 4-stage trainer')
958     parser.add_argument('--mode', choices=_all_modes, default='2hr',
959                         help='Training mode: sanity (≤10 min code check),
960
961                             'quick (≤10 min no-error), mini (≤1 hr
962                             learning check), '
963                             '2hr, full_train (~16-20 hr deployable
964                             model), '
965                             'full (alias for full_train), '
966                             'real_world (online robot fine-tuning). '
967                             'Aliases: smoke_test=quick,
968                             mini_train=mini.')
969     parser.add_argument('--scenario',
970                         choices=['warehouse', 'sar', 'maritime',
971                                 'search_and_rescue',
972                                 'maritime_security', 'all'],
973                         default='all',
974                         help='Run only this scenario (default: all)')

```

```

970 parser.add_argument('--workers', type=int, default=None,
971                     help='Override parallel workers')
972 parser.add_argument('--rl_episodes', type=int, default=None,
973                     help='Override RL episodes per job')
974 parser.add_argument('--diff_epochs', type=int, default=None,
975                     help='Override diffusion epochs per job')
976 parser.add_argument('--rl_only', action='store_true',
977                     help='Stop after Stage B (RL only; skip
diffusion)')
978 parser.add_argument('--stop_after', choices=['A', 'B', 'C', 'D'],
979                     default='D',
980                     help='Stop after this stage (default: D = full
pipeline)')
981 # run_fast.py behaviour, baked in
982 parser.add_argument('--no_amp', action='store_true',
983                     help='Disable AMP mixed precision (also disables
Min-SNR)')
984 parser.add_argument('--rl_batch', type=int, default=None,
985                     help='Override RL batch size via
AQUA_RL_BATCH_SIZE env var')
986 parser.add_argument('--diff_batch', type=int, default=None,
987                     help='Override diffusion batch size via
AQUA_DIFF_BATCH_SIZE')
988 parser.add_argument('--safe', action='store_true',
989                     help='Conservative batches: 256 / 128 instead of
512 / 256')
990 parser.add_argument('--gpu_lock_pct', type=float, default=0.50,
991                     help='Fraction of per-worker GPU cap to
physically lock '
992                     '(default: 0.50 = run_fast.py behaviour). '
993                     'Pass 0 to disable.')
994 parser.add_argument('--gpu_reserve_gb', type=float, default=None,

```

```

995             help='Total GPU VRAM budget (GB) split across
workers. '
996             'Default: auto-detect (free_gb - headroom).')
997 parser.add_argument('--gpu_headroom_gb', type=float, default=2.0,
998             help='GB of VRAM to leave free (default: 2.0)')
999 parser.add_argument('--gpu_full', action='store_true',
1000             help='Do not cap per-worker VRAM (risks OOM on
shared GPU)')
1001 parser.add_argument('--resume', type=str, default=None,
metavar='CHECKPOINT_DIR',
1002             help='Resume interrupted training from this
checkpoint directory. '
1003             'Completed stages are detected via stage pkl
files and skipped. '
1004             'Model weights are loaded from
CHECKPOINT_DIR if provided.')
1005
1006 # -----
1007 # Real-world online learning (entirely separate path; offline
1008 # training pipeline is not touched when this flag is present)
1009 # -----
1010 parser.add_argument('--real-world', action='store_true',
1011             help='Run online RL on a physical robot instead
of '
1012             'simulation. Requires --rw-scenario and
--rw-algo. '
1013             'All offline pipeline stages are skipped.')
1014 parser.add_argument('--rw-scenario',
1015             choices=['warehouse', 'sar', 'maritime',
1016             'search_and_rescue',
1017             'maritime_security'],
1018             default=None,
1019             help='Scenario for real-world training')
1020 parser.add_argument('--rw-algo',

```

```

1020             choices=['MAPPO', 'QMIX', 'MADDPG', 'DQN',
1021                     'HybridRL'],
1022             default='HybridRL',
1023             help='RL algorithm for real-world online
1024             training')
1025         parser.add_argument('--rw-checkpoint', type=str, default=None,
1026                             help='Path to pretrained .pt checkpoint to
1027                             warm-start '
1028                                 'from (recommended; robot starts with
1029                                 sim-trained '
1030                                 'behaviour and fine-tunes on real
1031                                 experience)')
1032         parser.add_argument('--rw-steps', type=int, default=500,
1033                             help='Number of episodes for online training
1034                             (default 500)')
1035         parser.add_argument('--rw-step-dt', type=float, default=0.1,
1036                             help='Real-time seconds per policy step (default
1037                             0.1)')
1038         parser.add_argument('--rw-async', action='store_true',
1039                             help='Use async actor/learner threads (no
1040                             episode-boundary '
1041                                 'pauses; required for fast-moving
1042                                 scenarios)')
1043         parser.add_argument('--rw-update-every', type=int, default=50,
1044                             help='Steps between gradient updates in sync
1045                             mode')
1046         parser.add_argument('--rw-save-every', type=int, default=50,
1047                             help='Episodes between checkpoint saves during
1048                             online training')
1049
1050     if args is None:
1051         args = parser.parse_args()

```



```

1042     # Resolve mode aliases (smoke_test→quick, mini_train→mini,
        full_train→full)
1043     args.mode = _MODE_ALIASES.get(args.mode, args.mode)
1044
1045     # Resolve --scenario to canonical internal names
1046     if args.scenario == 'all' or args.scenario is None:
1047         args.active_scenarios = SCENARIOS
1048     else:
1049         canonical = _SCENARIO_MAP.get(args.scenario)
1050         if canonical is None:
1051             parser.error(f'Unknown scenario: {args.scenario}')
1052         args.active_scenarios = [canonical]
1053
1054     # Set defaults for optional attributes that job_runner may not include
1055     for _attr, _default in [
1056         ('safe', False), ('no_amp', False), ('rl_batch', None),
1057         ('diff_batch', None),
1058         ('rl_episodes', None), ('diff_epochs', None), ('rl_only', False),
1059         ('stop_after', 'D'), ('gpu_lock_pct', 0.5), ('gpu_reserve_gb',
1060         None),
1061         ('gpu_headroom_gb', 2.0), ('gpu_full', False),
1062         ('real_world', False), ('rw_scenario', None), ('rw_algo',
1063         'HybridRL'),
1064         ('rw_checkpoint', None), ('rw_steps', 500), ('rw_step_dt', 0.1),
1065         ('rw_async', False), ('rw_update_every', 50), ('rw_save_every',
1066         50),
1067     ]:
1068         if not hasattr(args, _attr):
1069             setattr(args, _attr, _default)
1070
1071     # --safe: drop batches to the conservative defaults
1072     if args.safe:
1073         if args.rl_batch is None:
1074             args.rl_batch = 256

```

```

1071         if args.diff_batch is None:
1072             args.diff_batch = 128
1073
1074         # Spawn once at top level; enforce expandable_segments
1075         try:
1076             mp.set_start_method('spawn', force=True)
1077         except RuntimeError:
1078             pass
1079         os.environ.setdefault('PYTORCH_CUDA_ALLOC_CONF',
1080                               'expandable_segments:True')
1081
1082         mode_cfg = apply_mode(args.mode)
1083
1084         logger = setup_logging(args.mode)
1085         for d in (RESULTS_DIR, MODELS_DIR,
1086                 os.path.join(MODELS_DIR, 'checkpoints'), SIM_READY_DIR):
1087             os.makedirs(d, exist_ok=True)
1088
1089         gpu = torch.cuda.get_device_name(0) if torch.cuda.is_available() else
1090             'CPU'
1091
1092         # Resume: set checkpoint dir from --resume if provided
1093         if args.resume:
1094             resume_path = Path(args.resume)
1095             if not resume_path.exists():
1096                 logger.error(f'--resume path does not exist: {args.resume}')
1097                 sys.exit(1)
1098             os.environ['AQUA_RESUME_CKPT_DIR'] = str(resume_path)
1099             logger.info(f'Resuming from checkpoint directory: {resume_path}')
1100
1101         n_rl_ep = args.rl_episodes or mode_cfg['rl_episodes']
1102         n_diff_ep = args.diff_epochs or mode_cfg['diff_epochs']
1103         logger.info('=' * 80)
1104         logger.info(f'AQUA Guardian | mode={args.mode} ')

```

```

1103         f'scenario={getattr(args, "scenario", "all")}
        seed={SEED}')
1104     logger.info(f' GPU           : {gpu}')
1105     logger.info(f' Log file      : {os.path.join(LOGS_DIR,
        f"run_{args.mode}*.log")}')
1106     logger.info(f' Stage A (RL) : {N_STAGE_A} jobs ',
1107                 f'({n_rl_ep} ep x {mode_cfg["max_steps"]} steps/ep)')
1108     logger.info(f' Stage B (RL) : {N_STAGE_B} jobs ',
1109                 f'(HybridRL, mandatory distill from Stage A ckpts)')
1110     logger.info(f' Stage C (Dif): {N_STAGE_C} jobs ',
1111                 f'({n_diff_ep} ep x {mode_cfg["diff_collect"]} collect,
        on {DIFF_RL_PAIR})')
1112     logger.info(f' Stage D (Dif): {N_STAGE_D} jobs ',
1113                 f'(HybridDiffusion, mandatory distill from Stage C
        ckpts)')
1114     logger.info(f' Workers      : {args.workers or
        mode_cfg["num_workers"]}')
1115     logger.info(f' AMP          : {not args.no_amp}')
1116     logger.info(f' RL batch     : {args.rl_batch or
        mode_cfg["rl_batch_size"]}')
1117     logger.info(f' Diff batch    : {args.diff_batch or
        mode_cfg["diff_batch_size"]}')
1118     logger.info(f' Scenarios     : {getattr(args, "active_scenarios",
        SCENARIOS)}')
1119
1120     # GPU memory planning (shared-GPU safety)
1121     planned_workers = args.workers or mode_cfg['num_workers']
1122     args.per_worker_gb = None
1123     if args.gpu_full:
1124         logger.info(f' GPU budget    : --gpu_full (no cap; may OOM on
        shared GPU)')
1125     else:
1126         from utils.gpu_utils import compute_worker_budget,
        query_free_gpu_gb

```

```

1127     info = query_free_gpu_gb()
1128     plan = compute_worker_budget(
1129         num_workers=planned_workers,
1130         reserve_gb=args.gpu_reserve_gb,
1131         headroom_gb=args.gpu_headroom_gb,
1132     )
1133     logger.info(f' GPU VRAM      : {info["free_gb"]:.1f} GB free / '
1134               f'{info["total_gb"]:.1f} GB total')
1135     if plan['total_gb'] > 0:
1136         if plan['num_workers'] != planned_workers:
1137             logger.warning(
1138                 f' GPU plan      : reducing workers '
1139                 f'{planned_workers} -> {plan["num_workers"]} '
1140                 f'(budget was only '
1141                 f'{plan["total_budget_gb"]/planned_workers:.1f} GB /
worker)')
1142             planned_workers = plan['num_workers']
1143             args.workers = planned_workers
1144             logger.info(f' GPU plan      : {plan["per_worker_gb"]:.1f} GB
/ worker '
1145                       f'x {plan["num_workers"]} workers '
1146                       f'={plan["total_budget_gb"]:.1f} GB total '
1147                       f'(headroom {args.gpu_headroom_gb:.1f} GB)')
1148             args.per_worker_gb = float(plan['per_worker_gb'])
1149
1150     logger.info(f' Workers      : {planned_workers}')
1151     logger.info('=' * 80)
1152
1153     # -----
1154     # Real-world path: --mode real_world OR --real-world flag both route
1155     # here. The staged offline pipeline is not used in this path.
1156     # -----
1157     if args.mode == 'real_world' or getattr(args, 'real_world', False):
1158         _run_real_world(args, logger)

```

```

1159         return
1160
1161     t0 = time.time()
1162     results = run_staged(args.mode, mode_cfg, args, logger)
1163
1164     # Sim-ready export
1165     logger.info('\n>>> Exporting sim-ready .pt files')
1166     export_all(logger)
1167
1168     # Excel / stats. statistical_eval rewritten for the 15-diff grid.
1169     # No try/except wrapper (per no-fallbacks) - if export fails, we want
1170     to
1171     # know about it rather than silently finish.
1172     from training.statistical_eval import export_results_to_excel
1173     export_results_to_excel(results, RESULTS_DIR)
1174     logger.info(f' Excel: {RESULTS_DIR}/training_results.xlsx')
1175
1176     # Figures
1177     from training.visualizer import generate_all_figures
1178     fig_dir = os.path.join(RESULTS_DIR, 'figures')
1179     generated = generate_all_figures(RESULTS_DIR, fig_dir)
1180     logger.info(f' Figures: {len(generated)} -> {fig_dir}')
1181
1182     summarize(results, logger, args.mode)
1183     logger.info(f'\nDone in {(time.time()-t0)/60:.1f} min '
1184               f'({(time.time()-t0)/3600:.2f} h)')
1185
1186 if __name__ == '__main__':
1187     main()

```

Listing B.32: Main training orchestrator. Implements the four-stage barrier pipeline, parses command-line flags and dispatches jobs to the multiprocessing runner.

```

1  """
2  sanity.py - unified sanity / diagnostic script for AQUA Guardian.
3
4  68 checks covering the full spec verification matrix (K1--K64 + K65--K68):
5
6  python sanity.py                # all 68 checks
7  python sanity.py --quick         # skip slow checks (check 8)
8  python sanity.py --check 12     # single check by number
9  python sanity.py --scenario maritime_security
10 python sanity.py --timeout 60    # 60-second wall-clock limit per
    check
11
12 Check groups
13 =====
14 1--8   : RL & env core (smoke, robots, movement, bounds, completion,
15         reward decomp, distill, MAPPO smoke training)
16 9--11  : Diffusion models (instantiation, TDM loss descent, prefix
17         matching)
18 12--18 : Task 4-state status machine (K8--K14)
19 19--21 : Battery drain and robot deactivation (K18--K20)
20 22--30 : Hexagonal grid utilities (K21--K29)
21 31--33 : Action masking (K34--K36)
22 34--35 : PCGrad gradient surgery (K37--K38)
23 36--38 : QMIX dynamic fleet + ReplayBuffer (K39--K41)
24 39--44 : Diffusion pipeline details (K46--K51)
25 45--51 : Task type tables and DAG invariants (K52--K58)
26 52--55 : Scalable policy --- MAPPO/QMIX for N=2 and N=20 (K61--K64)
27 56--64 : Scenario smokes (SAR, maritime), finite rewards, mode aliases,
28         SimReadyRL wrapper, obs bounds, masking correctness
29 65--68 : New feature checks --- MODES dict, PerformancePacedGenerator,
30         scenario config grid sizes, scenario-aware masking params
31
32 Each check returns PASS / FAIL / AMBIG. The script aggregates and
    prints a summary with per-check timing at the end.

```

```

33 """
34
35 from __future__ import annotations
36
37 import argparse
38 import concurrent.futures
39 import inspect
40 import os
41 import re
42 import sys
43 import time
44 import traceback
45 from collections import defaultdict
46 from pathlib import Path
47 from typing import Dict, List, Optional, Tuple
48
49 import numpy as np
50 import torch
51
52
53 # Project import bootstrap
54
55 def _bootstrap() -> None:
56     here = Path(__file__).resolve().parent
57     candidates = [here, here.parent, Path.cwd(), Path.cwd() /
58                   'aqua_guardian']
59     for c in candidates:
60         if (c / 'config' / 'base_config.py').is_file() and\
61             (c / 'environment' / 'task_env.py').is_file():
62             if str(c) not in sys.path:
63                 sys.path.insert(0, str(c))
64             os.chdir(c)
65             return
66     sys.stderr.write('[sanity] cannot find aqua_guardian layout\n')

```

```

66     sys.exit(2)
67
68
69 _bootstrap()
70
71 from config.base_config import (                                # noqa: E402
72     MAX_ROBOTS, MAX_TASKS,
73     LOCAL_OBS_DIM, GLOBAL_OBS_DIM, FLAT_OBS_DIM,
74     ACTION_DIM, SCENARIO_CONFIGS,
75     ROBOT_STATUS_ACTIVE, ROBOT_STATUS_INACTIVE,
76 )
77 from environment.task_env import ImprovedTaskAllocationEnv # noqa: E402
78
79
80 # Print helpers
81
82 _BOLD = '\033[1m'; _RED = '\033[91m'; _GREEN = '\033[92m'
83 _YEL = '\033[93m'; _BLUE = '\033[94m'; _END = '\033[0m'
84 if not sys.stdout.isatty():
85     _BOLD = _RED = _GREEN = _YEL = _BLUE = _END = ''
86
87
88 def _hdr(t: str) -> None:
89     bar = '-' * 78
90     print(f'\n{_BOLD}{_BLUE}{bar}\n {t}\n{bar}{_END}', flush=True)
91
92
93 def _v(label: str, kind: str, detail: str = '') -> None:
94     color = {'PASS': _GREEN, 'FAIL': _RED, 'AMBIG': _YEL,
95             'INFO': _BLUE}.get(kind, '')
96     head = f'{_BOLD}{color}[{kind:>5}]{_END} {label}'
97     if detail:
98         head = head + ' --- ' + detail
99     print(head, flush=True)

```



```

100
101
102 # Shared minimal env builder
103
104 def _build_env(scenario: str, num_robots: int, num_tasks: int,
105               max_steps: int):
106     env = ImprovedTaskAllocationEnv(scenario=scenario)
107     env.config.MAX_STEPS_PER_EPISODE = int(max_steps)
108     _orig_reset = env.reset
109
110     def _patched_reset(seed=None, options=None):
111         _orig_reset(seed=seed, options=options)
112         env.num_robots = int(num_robots)
113         env.num_tasks = int(num_tasks)
114         env.robots = env._generate_robots()
115         env.tasks = env._generate_tasks()
116         for r in env.robots:
117             r['operational_status'] = ROBOT_STATUS_ACTIVE
118         env.task_assignments = {}
119         env.robot_loads = {i: 0 for i in range(env.num_robots)}
120         env.completed_tasks = set()
121         env._rewarded_task_ids = set()
122         env.current_step = 0
123         env.robot_targets = {i: None for i in range(env.num_robots)}
124         env.task_progress = {i: 0.0 for i in range(env.num_tasks)}
125         env._robot_prev_pos = {}
126         env._robot_travel_dist = {}
127         env._robot_waiting_time = {}
128         return env._get_observation(), env._get_info()
129
130     env.reset = _patched_reset
131     return env
132
133

```

```

134 # CHECK 1 - env smoke
135
136 def check_1(scenario: str = 'warehouse') -> Dict:
137     _hdr(f'CHECK 1: env smoke ({scenario})')
138     try:
139         env = ImprovedTaskAllocationEnv(scenario=scenario)
140         obs, info = env.reset(seed=42)
141         no_op = env.max_robots * env.max_tasks
142         obs2, r, term, trunc, info2 = env.step(no_op)
143     except Exception as e:
144         _v('CHECK 1', 'FAIL', f'{type(e).__name__}: {e}')
145         traceback.print_exc()
146         return {'verdict': 'FAIL', 'error': str(e)}
147
148     print(f'  obs shape          : {obs.shape}')
149     print(f'  expected            : ({MAX_ROBOTS*15 + MAX_TASKS*12 + 10},)')
150     print(f'  num_robots           : {env.num_robots}')
151     print(f'  num_tasks            : {env.num_tasks}')
152     print(f'  step reward          : {r:+.3f}')
153     sc_cfg = SCENARIO_CONFIGS[scenario]
154     print(f'  scenario default : num_robots={sc_cfg.num_robots}')
155     if obs.shape == obs2.shape and len(obs) > 0:
156         _v('CHECK 1', 'PASS',
157             f'env instantiates, resets, and steps cleanly')
158         return {'verdict': 'PASS'}
159     _v('CHECK 1', 'FAIL', 'observation shapes inconsistent')
160     return {'verdict': 'FAIL'}
161
162
163 # CHECK 2 - operational_status='active'
164
165 def check_2(scenario: str = 'warehouse') -> Dict:
166     _hdr(f'CHECK 2: spawned robots have status=\"active\" ({scenario})')
167     env = ImprovedTaskAllocationEnv(scenario=scenario)

```

```

168     env.reset(seed=42)
169     counts = defaultdict(int)
170     for r in env.robots:
171         counts[r.get('operational_status', '<missing>')] += 1
172     print(f'    spawned robots    : {len(env.robots)}')
173     for s, n in sorted(counts.items(), key=lambda kv: -kv[1]):
174         print(f'        {s!r:<20} : {n}')
175     n_active = counts.get('active', 0)
176     if n_active == len(env.robots) and n_active > 0:
177         _v('CHECK 2', 'PASS', f'all {n_active} robots are active')
178         return {'verdict': 'PASS', 'active': n_active}
179     _v('CHECK 2', 'FAIL',
180        f'{n_active}/{len(env.robots)} robots are active')
181     return {'verdict': 'FAIL', 'active': n_active}
182
183
184 # CHECK 3 - movement integration
185
186 def check_3(scenario: str = 'warehouse', n_steps: int = 200) -> Dict:
187     _hdr(f'CHECK 3: _update_robot_positions moves a robot ({scenario})')
188     env = _build_env(scenario, num_robots=1, num_tasks=1, max_steps=1000)
189     env.reset(seed=42)
190     half = float(min(env.environment['grid_size']) / 2.0)
191     target = (half * 0.5, 0.0)
192     env.robots[0].update({
193         'position': [0.0, 0.0], 'velocity': [0.0, 0.0],
194         'operational_status': ROBOT_STATUS_ACTIVE, 'current_payload': 0.0,
195         'current_battery': env.robots[0]['battery_capacity'],
196     })
197     env.tasks[0].update({
198         'location': list(target), 'status': 'pending',
199         'estimated_duration': 1.0, 'dependencies': [],
200         'payload_weight': 1.0,
201     })

```

```

202     env.robot_targets[0] = 0
203     env.robots[0]['current_task'] = 0
204     env.task_progress[0] = 0.0
205     env.tasks[0]['status'] = 'assigned'
206
207     dt = env.dataset_config.DT
208     distances: List[float] = []
209     for _ in range(n_steps):
210         try:
211             env._update_robot_positions(dt)
212         except Exception as e:
213             _v('CHECK 3', 'FAIL', f'{type(e).__name__}: {e}')
214             return {'verdict': 'FAIL', 'error': str(e)}
215         rx, ry = env.robots[0]['position'][0],
env.robots[0]['position'][1]
216         d = float(np.hypot(rx - target[0], ry - target[1]))
217         distances.append(d)
218         if 0 in env.completed_tasks:
219             break
220     moved = float(np.hypot(env.robots[0]['position'][0],
env.robots[0]['position'][1]))
221
222     print(f' dt : {dt}')
223     print(f' steps simulated : {len(distances)}')
224     print(f' total displacement: {moved:.3f} m')
225     print(f' final distance : {distances[-1]:.3f} m '
f'(arrival_tol={env._arrival_tol:.2f})')
226
227     if moved < 0.01:
228         _v('CHECK 3', 'FAIL', 'robot did not move')
229         return {'verdict': 'FAIL', 'displacement': moved}
230     if distances[-1] <= env._arrival_tol:
231         _v('CHECK 3', 'PASS',
f'arrived in {len(distances)} steps')
232         return {'verdict': 'PASS', 'displacement': moved}
233     _v('CHECK 3', 'AMBIG',

```

```

235         'moved but did not arrive --- speed/dt may be off')
236     return {'verdict': 'AMBIG', 'displacement': moved}
237
238
239 # CHECK 4 - defensive bounds
240
241 def check_4(scenario: str = 'warehouse') -> Dict:
242     _hdr(f'CHECK 4: stale target_task_id does not crash ({scenario})')
243     env = _build_env(scenario, num_robots=2, num_tasks=3, max_steps=10)
244     env.reset(seed=42)
245     env.robot_targets[0] = 999
246     env.robots[0]['operational_status'] = ROBOT_STATUS_ACTIVE
247     env.robots[0]['current_task'] = 999
248     no_op = env.max_robots * env.max_tasks
249     crashed = False
250     err = ''
251     try:
252         env.step(no_op)
253     except Exception as e:
254         crashed = True
255         err = f'{type(e).__name__}: {e}'
256     print(f' set robot_targets[0] = 999 (num_tasks={env.num_tasks})')
257     print(f' step crashed      : {crashed}')
258     if crashed:
259         print(f' error              : {err}')
260         _v('CHECK 4', 'FAIL', 'env still crashes on stale target')
261         return {'verdict': 'FAIL', 'error': err}
262     _v('CHECK 4', 'PASS', 'env handles stale target defensively')
263     return {'verdict': 'PASS'}
264
265
266 # CHECK 5 - end-to-end completion via env.step
267
268 def check_5(scenario: str = 'warehouse', max_steps: int = 2000) -> Dict:

```

```

269     _hdr(f'CHECK 5: end-to-end one-robot one-task completion
        ({scenario}))')
270     env = _build_env(scenario, num_robots=1, num_tasks=1,
        max_steps=max_steps)
271     env.reset(seed=42)
272     half = float(min(env.environment['grid_size']) / 2.0)
273     target = (half * 0.5, 0.0)
274     env.robots[0].update({
275         'position': [0.0, 0.0], 'velocity': [0.0, 0.0],
276         'operational_status': ROBOT_STATUS_ACTIVE, 'current_task': None,
277         'current_battery': env.robots[0]['battery_capacity'],
278         'current_payload': 0.0,
279     })
280     env.tasks[0].update({
281         'location': list(target), 'status': 'AVAILABLE',
282         'assigned_robot': None,
283         'payload_weight': min(env.tasks[0].get('payload_weight', 1.0),
284                               max(1.0, env.robots[0]['max_payload'] *
285                                   0.5)),
286         'estimated_duration': 1.0, 'dependencies': [], 'priority': 'high',
287     })
288     env.task_progress[0] = 0.0
289     env.dynamic_obstacles = []
290     if 'static_obstacles' in env.environment:
291         env.environment['static_obstacles'] = []
292
293     assign = 0 * env.max_tasks + 0
294     no_op = env.max_robots * env.max_tasks
295     arrived_step: Optional[int] = None
296     completed_step: Optional[int] = None
297     cum_r = 0.0
298     distances: List[float] = []
299     for step_n in range(max_steps):
300         try:

```

```

300         _, r, term, trunc, _ = env.step(assign if step_n == 0 else
no_op)
301     except Exception as e:
302         _v('CHECK 5', 'FAIL',
303             f'step crashed at {step_n}: {type(e).__name__}: {e}')
304         return {'verdict': 'FAIL', 'error': str(e)}
305     cum_r += r
306     rx, ry = env.robots[0]['position'][0],
env.robots[0]['position'][1]
307     d = float(np.hypot(rx - target[0], ry - target[1]))
308     distances.append(d)
309     if arrived_step is None and d <= env._arrival_tol:
310         arrived_step = step_n
311     if 0 in env.completed_tasks and completed_step is None:
312         completed_step = step_n
313         break
314     if term or trunc:
315         break
316
317     print(f'  steps run          : {len(distances)}')
318     print(f'  initial distance : {np.hypot(*target):.2f} m')
319     print(f'  min distance      : {min(distances):.2f} m    '
320         f'at step {int(np.argmin(distances))}')
321     print(f'  arrived (tol={env._arrival_tol:.2f}) at: {arrived_step}')
322     print(f'  completed step    : {completed_step}')
323     print(f'  cumulative reward: {cum_r:+.2f}')
324
325     if completed_step is not None:
326         _v('CHECK 5', 'PASS',
327             f'completed at step {completed_step} (arrival at
{arrived_step}'))
328         return {'verdict': 'PASS', 'arrived_step': arrived_step,
329                 'completed_step': completed_step}
330     if arrived_step is not None:

```

```

331     _v('CHECK 5', 'AMBIG',
332         f'arrived at step {arrived_step} but task did not complete; '
333         'check task_progress / estimated_duration')
334     return {'verdict': 'AMBIG', 'arrived_step': arrived_step}
335     _v('CHECK 5', 'FAIL',
336         f'never reached target (min dist {min(distances):.2f} m)')
337     return {'verdict': 'FAIL', 'min_distance': min(distances)}
338
339
340 # CHECK 6 - reward decomposition
341
342 def check_6(scenario: str = 'warehouse', num_robots: int = 5,
343             max_steps: int = 500) -> Dict:
344     _hdr(f'CHECK 6: reward decomposition ({scenario}, '
345         f'random-valid policy, {num_robots} robots, {max_steps} steps)')
346     env = _build_env(scenario, num_robots=num_robots,
347                     num_tasks=10, max_steps=max_steps)
348     env.reset(seed=42)
349     cfg = env.config
350     print(f'  reward constants:')
351     print(f'    REWARD_TASK_COMPLETION   :
352           {cfg.REWARD_TASK_COMPLETION:+.2f}')
353     print(f'    REWARD_PRIORITY_BONUS      :
354           {cfg.REWARD_PRIORITY_BONUS:+.2f}')
355     print(f'    REWARD_EFFICIENCY             : {cfg.REWARD_EFFICIENCY:+.2f}')
356     print(f'    PENALTY_INCOMPLETE             : {cfg.PENALTY_INCOMPLETE:+.2f}')
357     print(f'    PENALTY_COLLISION               : {cfg.PENALTY_COLLISION:+.2f}')
358     print(f'    PENALTY_ROBOT_COLLISION         :
359           {cfg.PENALTY_ROBOT_COLLISION:+.2f}')
360     print(f'    no-op penalty                   : -0.10')
361     print(f'    MAX_STEPS_PER_EPISODE           : {cfg.MAX_STEPS_PER_EPISODE}')
362
363     buckets: Dict[str, float] = defaultdict(float)
364     counts: Dict[str, int] = defaultdict(int)

```



```

362     _orig_step = env.step
363
364     def _wrap(action: int):
365         before_completed = set(env.completed_tasks)
366         before_assigned = set(env.task_assignments.keys())
367         before_obs_col = env.collisions
368         before_rob_col = env.robot_collisions
369         obs, total_r, term, trunc, info = _orig_step(action)
370
371         if action == env.max_robots * env.max_tasks:
372             buckets['noop_penalty'] += -0.1
373             counts['noop_action'] += 1
374         else:
375             r_id = action // env.max_tasks
376             t_id = action % env.max_tasks
377             valid = (r_id < env.num_robots and t_id < env.num_tasks
378                     and t_id not in before_completed
379                     and t_id not in before_assigned)
380             if not valid:
381                 buckets['invalid_action'] += cfg.PENALTY_INCOMPLETE
382                 counts['invalid_action'] += 1
383             else:
384                 if t_id in env.task_assignments and t_id not in
before_assigned:
385                     pri = env.tasks[t_id].get('priority', 'low')
386                     pri_v = env.priority_encoding.get(pri, 1) / 3.0
387                     buckets['priority_bonus'] +=
cfg.REWARD_PRIORITY_BONUS * pri_v
388                     counts['successful_assign'] += 1
389                 else:
390                     buckets['capacity_or_dep'] += cfg.PENALTY_INCOMPLETE
* 0.5
391                     counts['capacity_or_dep'] += 1
392

```

```

393         if env.collisions - before_obs_col > 0:
394             buckets['obstacle_collision'] += cfg.PENALTY_COLLISION
395             counts['obstacle_collision_steps'] += 1
396         if env.robot_collisions - before_rob_col > 0:
397             buckets['robot_collision'] += cfg.PENALTY_ROBOT_COLLISION
398             counts['robot_collision_steps'] += 1
399
400         new_completed = env.completed_tasks - before_completed
401         if new_completed:
402             buckets['task_completion'] += cfg.REWARD_TASK_COMPLETION *
len(new_completed)
403             counts['task_completions'] += len(new_completed)
404             buckets['efficiency_estimate'] += cfg.REWARD EFFICIENCY * 0.5
* len(new_completed)
405
406         if term:
407             buckets['terminal_bonus'] += cfg.REWARD_TASK_COMPLETION * 2
408             counts['terminal'] += 1
409
410         buckets['_actual_total'] += total_r
411         return obs, total_r, term, trunc, info
412
413     env.step = _wrap
414     rng = np.random.RandomState(42)
415     done = False
416     n = 0
417     while not done and n < max_steps:
418         valid = []
419         for ri in range(env.num_robots):
420             for ti in range(env.num_tasks):
421                 if ti not in env.completed_tasks and ti not in
env.task_assignments:
422                     valid.append(ri * env.max_tasks + ti)
423                 a = int(rng.choice(valid)) if valid else env.max_robots *

```

```

env.max_tasks
424     _, _, term, trunc, _ = env.step(a)
425     done = term or trunc
426     n += 1
427
428     actual_total = buckets.pop('_actual_total', 0.0)
429     print()
430     print(f'  steps run          : {n}')
431     print(f'  task completions : {counts.get("task_completions", 0)}/'
432           f'{env.num_tasks}')
433     print(f'  successful assigns: {counts.get("successful_assign", 0)}')
434     print(f'  invalid actions  : {counts.get("invalid_action", 0)}')
435     print(f'  obstacle col steps: {counts.get("obstacle_collision_steps",
436                                     0)}')
437     print(f'  robot col steps  : {counts.get("robot_collision_steps",
438                                     0)}')
439     print()
440     print('  REWARD BUDGET:')
441     items = sorted(buckets.items(), key=lambda kv: kv[1])
442     width = max(len(k) for k, _ in items) if items else 20
443     for k, v in items:
444         c = _RED if v < 0 else _GREEN
445         print(f'      {k:<{width}} : {c}{v:+10.2f}{_END}')
446     print(f'      {"actual_total":<{width}} : '
447           f'{actual_total:+10.2f} (env-emitted)')
448
449     if counts.get('task_completions', 0) == 0:
450         worst = min(items, key=lambda kv: kv[1])
451         _v('CHECK 6', 'FAIL',
452            f'no completions; dominant penalty: {worst[0]}
453            ({worst[1]:+.2f})')
454
455         return {'verdict': 'FAIL', 'buckets': dict(buckets)}
456
457     pos = sum(v for _, v in items if v > 0)
458     neg = sum(v for _, v in items if v < 0)

```

```

454     if pos + neg > 0:
455         _v('CHECK 6', 'PASS',
456             f'{counts["task_completions"]} completions, net
+{pos+neg:.2f}')
457         return {'verdict': 'PASS', 'buckets': dict(buckets)}
458     _v('CHECK 6', 'AMBIG',
459         f'{counts["task_completions"]} completions but penalties dominate')
460     return {'verdict': 'AMBIG', 'buckets': dict(buckets)}
461
462
463 # CHECK 7 - distill diagnosis (imports run.py)
464
465 def check_7() -> Dict:
466     _hdr('CHECK 7: HybridRL distill warm-starts > 0 tensors')
467     from models.rl_agents import (
468         MAPPO, QMIX, MADDPG, DQN, HybridRL,
469     )
470     hybrid = HybridRL()
471     target_sd = hybrid.state_dict()
472
473     SUB_MAP = {
474         'MAPPO': ['mappo_actor', 'mappo_critic'],
475         'QMIX': ['qmix_agent'],
476         'MADDPG': ['maddpg_actor'],
477         'DQN': ['dqn_encoder', 'dqn_head'],
478     }
479     BASE = {
480         'MAPPO': MAPPO(), 'QMIX': QMIX(),
481         'MADDPG': MADDPG(), 'DQN': DQN(),
482     }
483
484     # Read run.py source to detect which distill version is currently in
+place
485     run_src = Path('run.py').read_text()

```

```

486     has_v5_fix = ('SENTINEL: distill-strip-v5' in run_src
487                  or 'SENTINEL: distill-strip-first-component' in run_src
488                  or 'stripped = k.split' in run_src)
489     print(f' run.py distill sentinel detected: {has_v5_fix}')
490
491     total_keys = 0
492     total_match = 0
493     print()
494     print(f' {"algo":<8}{ "base keys":<12}{ "matched":<10}{ "%
495           matched":<10}\'')
496     for algo, prefixes in SUB_MAP.items():
497         sd = BASE[algo].state_dict()
498         n_match = 0
499         for k, v in sd.items():
500             stripped = k.split('.', 1)[1] if has_v5_fix and '.' in k else
501             k
502             for pfx in prefixes:
503                 if has_v5_fix:
504                     tgt = f'{pfx}.{stripped}'
505                 else:
506                     tgt = f'{pfx}.{k}' if not k.startswith(pfx) else k
507                 if tgt in target_sd and target_sd[tgt].shape == v.shape:
508                     n_match += 1
509                 break
510         n = len(sd)
511         pct = (100.0 * n_match / n) if n else 0.0
512         c = _GREEN if n_match == n else (_YEL if n_match > 0 else _RED)
513         print(f' {c}{algo:<8}{n:<12}{n_match:<10}{pct:>6.1f}%{_END}\'')
514         total_keys += n
515         total_match += n_match
516
517     pct_total = (100.0 * total_match / total_keys) if total_keys else 0.0
518     print(f'\n TOTAL: {total_match}/{total_keys} ({pct_total:.1f}%)')

```

```

518     if total_match >= total_keys * 0.25:
519         _v('CHECK 7', 'PASS',
520             f'{total_match}/{total_keys} matches --- distill is
meaningful')
521         return {'verdict': 'PASS', 'matched': total_match, 'total':
total_keys}
522     if total_match == 0:
523         _v('CHECK 7', 'FAIL',
524             '0 matches - distill sentinel not present in run.py')
525         return {'verdict': 'FAIL', 'matched': 0, 'total': total_keys}
526     _v('CHECK 7', 'AMBIG', f'{total_match}/{total_keys} matches ---
partial')
527     return {'verdict': 'AMBIG', 'matched': total_match, 'total':
total_keys}
528
529
530 # CHECK 8 - 1000-ep MAPPO smoke training
531
532 _TREND_RE = re.compile(
533
534     r'ep=\s*(\d+)\s+robots=\d+\s+reward=\s*([-\d.]+)\s+completion=\s*([\d.]+)%'
535 )
536
537 class _TrainerLogCapture:
538     """Captures ep=... lines from stdout to recover trend even when the
539     trainer's results dict is unreadable."""
540     def __init__(self):
541         self.records: List[Tuple[int, float, float]] = []
542         self._orig_write = sys.stdout.write
543
544     def __enter__(self):
545         def _w(s):
546             self._orig_write(s)

```

```

547         for line in s.splitlines():
548             m = _TREND_RE.search(line)
549             if m:
550                 self.records.append(
551                     (int(m.group(1)),
552                     float(m.group(2)),
553                     float(m.group(3))))
554             return len(s)
555         sys.stdout.write = _w
556         return self
557
558     def __exit__(self, *_):
559         sys.stdout.write = self._orig_write
560
561
562 def check_8(scenario: str = 'warehouse', num_episodes: int = 200,
563            max_steps: int = 100,
564            out_dir: str = 'results/sanity') -> Dict:
565     _hdr(f'CHECK 8: {num_episodes}-ep MAPPO smoke ({scenario})')
566     os.makedirs(out_dir, exist_ok=True)
567
568     from models.rl_agents import MAPPO
569     from config.base_config import RLConfig
570     sc_cfg = SCENARIO_CONFIGS[scenario]
571     print(f'  scenario num_robots: {sc_cfg.num_robots}')
572     print(f'  max_steps           : {max_steps}  '
573           '(v5: 200 default for ~10-min runtime; 500 was too slow)')
574
575     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
576     print(f'  device           : {device}')
577     os.environ.setdefault('AQUA_AMP', '1' if device.type == 'cuda' else
578                           '0')
579
580     agent = MAPPO()

```

```

580     env = ImprovedTaskAllocationEnv(scenario=scenario)
581     env.config.MAX_STEPS_PER_EPISODE = int(max_steps)
582
583     from training.rl_trainer import train_rl_agent
584     import training.rl_trainer as _rl_mod
585     _rl_mod.cfg.num_episodes = num_episodes
586     _rl_mod.cfg.max_steps     = max_steps
587
588     capture = _TrainerLogCapture()
589     t0 = time.time()
590     results = None
591     err = None
592     with capture:
593         try:
594             results = train_rl_agent(
595                 agent=agent, env=env, scenario=scenario, algo='MAPPO',
596                 seed=42, num_episodes=num_episodes, device=device,
597                 log_every=max(1, num_episodes // 40),
598             )
599         except Exception as e:
600             err = f'{type(e).__name__}: {e}'
601             traceback.print_exc()
602     wall = time.time() - t0
603
604     if err:
605         _v('CHECK 8', 'FAIL', f'trainer crashed: {err}')
606         return {'verdict': 'FAIL', 'error': err}
607
608     rewards: List[float] = []
609     completions: List[float] = []
610     if results:
611         rewards      = list(results.get('episode_rewards', []) or [])
612         completions  = list(results.get('completion_rates', []) or [])
613

```



```

614     if not rewards and capture.records:
615         rewards      = [r[1]      for r in capture.records]
616         completions = [r[2]/100 for r in capture.records]
617         print(f' (recovered {len(rewards)} log records from stdout)')
618
619     if not rewards:
620         _v('CHECK 8', 'FAIL',
621             'trainer returned no rewards and no log lines were captured')
622         return {'verdict': 'FAIL'}
623
624     early = rewards[: max(1, len(rewards) // 5)]
625     late  = rewards[-max(1, len(rewards) // 5):]
626     e_r = float(np.mean(early)); l_r = float(np.mean(late)); d_r = l_r -
        e_r
627     e_c = (float(np.mean(completions[: max(1, len(completions)//5)]))
628            if completions else float('nan'))
629     l_c = (float(np.mean(completions[-max(1, len(completions)//5):]))
630            if completions else float('nan'))
631
632     print()
633     print(f' episodes run      : {len(rewards)}')
634     print(f' wall time        : {wall/60:.1f} min')
635     print(f' early reward mean: {e_r:+8.2f} (first 20%)')
636     print(f' late reward mean : {l_r:+8.2f} (last 20%)')
637     print(f' Delta reward      : {d_r:+8.2f}')
638     print(f' early compl mean : {100*e_c:5.1f}%')
639     print(f' late compl mean  : {100*l_c:5.1f}%')
640
641     try:
642         import matplotlib
643         matplotlib.use('Agg')
644         import matplotlib.pyplot as plt
645         fig, ax = plt.subplots(1, 1, figsize=(9, 4.5))
646         ax.plot(rewards, alpha=0.35, lw=0.7, label='per-episode')

```

```

647     win = max(10, len(rewards) // 50)
648     if len(rewards) >= win:
649         sm = np.convolve(rewards, np.ones(win)/win, mode='valid')
650         ax.plot(np.arange(win-1, len(rewards)), sm, lw=1.8,
651                label=f'rolling-{win}')
652     ax.set_xlabel('episode'); ax.set_ylabel('total reward')
653     ax.set_title(f'CHECK 8 --- MAPPO/{scenario} '
654                 f'({sc_cfg.num_robots} robots, {num_episodes} ep)')
655     ax.legend(loc='best'); ax.grid(alpha=0.25)
656     out_png = os.path.join(out_dir, f'check8_{scenario}.png')
657     fig.tight_layout(); fig.savefig(out_png, dpi=110); plt.close(fig)
658     print(f'    saved plot          : {out_png}')
659 except Exception as e:
660     print(f'    (plot skipped: {e})')
661
662 if d_r >= 5.0 or (l_c - e_c) >= 0.05:
663     _v('CHECK 8', 'PASS',
664        f'measurable improvement (Delta={d_r:+.2f}, '
665        f'compl {100*e_c:.1f}%->{100*l_c:.1f}%)')
666     return {'verdict': 'PASS', 'delta': d_r,
667            'early_compl': e_c, 'late_compl': l_c}
668 if d_r >= 1.0:
669     _v('CHECK 8', 'AMBIG',
670        f'small improvement (Delta={d_r:+.2f}); '
671        'try longer run or hyperparam tuning')
672     return {'verdict': 'AMBIG', 'delta': d_r}
673 if d_r < -2.0:
674     _v('CHECK 8', 'FAIL',
675        f'reward got worse (Delta={d_r:+.2f}); algo unstable')
676     return {'verdict': 'FAIL', 'delta': d_r}
677 _v('CHECK 8', 'FAIL',
678    f'flat reward (Delta={d_r:+.2f}); env or hyperparam issue remains')
679 return {'verdict': 'FAIL', 'delta': d_r}
680

```

```

681
682 # CHECK 9 - diffusion model instantiation + forward pass
683
684 def check_9() -> Dict:
685     _hdr('CHECK 9: all 5 diffusion models instantiate and forward')
686     from models.diffusion_models import (
687         TDM, DiffusionPolicyModel, Diffuser, SDE, HybridDiffusion,
688     )
689     from config.base_config import GLOBAL_OBS_DIM
690     n_eval = 20 # bspline_n_eval
691
692     classes = [
693         ('TDM', TDM),
694         ('DiffusionPolicy', DiffusionPolicyModel),
695         ('Diffuser', Diffuser),
696         ('SDE', SDE),
697         ('HybridDiffusion', HybridDiffusion),
698     ]
699
700     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
701
702     obs = torch.randn(1, GLOBAL_OBS_DIM, device=device)
703     task_locs = torch.tensor([[0.5, 0.3], [0.1, -0.2]],
704                               dtype=torch.float32, device=device)
705     other_robot_locs = torch.zeros(0, 2, device=device)
706
707     print(f' device : {device}')
708     print(f' obs shape : {tuple(obs.shape)}')
709     print(f' task_locs shape : {tuple(task_locs.shape)}')
710     print(f' other_robot_locs : {tuple(other_robot_locs.shape)} (none)')
711     print()
712
713     failures: List[Tuple[str, str]] = []
714     for name, cls in classes:

```

```

715     try:
716         m = cls(obs_dim=GLOBAL_OBS_DIM).to(device).eval()
717         n_params = sum(p.numel() for p in m.parameters())
718         with torch.no_grad():
719             out = m(obs, task_locs, other_robot_locs)
720             if isinstance(out, tuple) and len(out) == 2:
721                 rid, tid = out
722                 shape_str = (f'(robot_id={int(rid.item())}, '
723                             f'task_id={int(tid.item())})')
724             elif torch.is_tensor(out):
725                 shape_str = str(tuple(out.shape))
726             else:
727                 shape_str = str(type(out).__name__)
728             _v(f' {name:<16}', 'PASS',
729               f'{n_params:>9,d} params, returns {shape_str}')
730     except Exception as e:
731         failures.append((name, f'{type(e).__name__}: {e}'))
732         _v(f' {name:<16}', 'FAIL', f'{type(e).__name__}: {e}')
733
734     if not failures:
735         _v('CHECK 9', 'PASS',
736           'all 5 diffusion models instantiate and run forward')
737         return {'verdict': 'PASS'}
738     _v('CHECK 9', 'FAIL',
739       f'{len(failures)} diffusion model(s) failed to instantiate or run')
740     return {'verdict': 'FAIL', 'failures': failures}
741
742
743     # CHECK 10 - diffusion training loss descends on synthetic data
744
745     def check_10(n_steps: int = 30) -> Dict:
746         _hdr(f'CHECK 10: TDM training loss descends on synthetic batch '
747             f'({n_steps} steps)')
748         from models.diffusion_models import TDM

```

```

749     from config.base_config import GLOBAL_OBS_DIM
750
751     n_eval = 20
752
753     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
754     batch = 16
755
756     torch.manual_seed(7)    # seed model init and training_loss noise
757                             samples
758     rng = torch.Generator(device=device).manual_seed(42)
759
760     obs      = torch.randn(batch, GLOBAL_OBS_DIM, generator=rng,
761                             device=device)
762
763     start_pos = torch.randn(batch, 2, generator=rng,
764                             device=device) * 0.1
765
766     goal_pos  = torch.randn(batch, 2, generator=rng,
767                             device=device) * 0.5
768
769     t_lin = torch.linspace(0.0, 1.0, n_eval, device=device) # (n_eval,)
770     t_lin = t_lin.unsqueeze(0).unsqueeze(-1)                # (1,
771                             n_eval, 1)
772
773     s_b    = start_pos.unsqueeze(1)                          # (batch, 1,
774                             2)
775
776     g_b    = goal_pos.unsqueeze(1)                           # (batch, 1,
777                             2)
778
779     traj_wpts = s_b + (g_b - s_b) * t_lin                    # (batch,
780                             n_eval, 2)
781
782     model = TDM(obs_dim=GLOBAL_OBS_DIM).to(device).train()
783     opt   = torch.optim.Adam(model.parameters(), lr=3e-3)
784
785     losses: List[float] = []
786     for step in range(n_steps):
787         opt.zero_grad()

```

```

775         try:
776             loss = model.training_loss(traj_wpts, obs, start_pos,
goal_pos)
777         except Exception as e:
778             _v('CHECK 10', 'FAIL',
779                 f'training_loss raised {type(e).__name__}: {e}')
780             return {'verdict': 'FAIL', 'error': str(e)}
781         loss.backward()
782         torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
783         opt.step()
784         losses.append(float(loss.detach()))
785
786     early = float(np.mean(losses[: max(1, n_steps // 5)]))
787     late  = float(np.mean(losses[-max(1, n_steps // 5):]))
788     drop  = early - late
789
790     print(f'  steps run      : {n_steps}')
791     print(f'  early mean    : {early:.4f}')
792     print(f'  late  mean     : {late:.4f}')
793     print(f'  drop           : {drop:+.4f}  '
794           f'want > 0; bigger is better')
795     print(f'  loss[0]         : {losses[0]:.4f}')
796     print(f'  loss[{n_steps-1}] : {losses[-1]:.4f}')
797
798     if drop > 0.05:
799         _v('CHECK 10', 'PASS',
800             f'loss descended by {drop:.4f}; gradient flow works')
801         return {'verdict': 'PASS', 'drop': drop, 'losses': losses}
802     if drop > 0:
803         _v('CHECK 10', 'AMBIG',
804             f'tiny drop ({drop:.4f}); may need more steps but trend is
right')
805         return {'verdict': 'AMBIG', 'drop': drop}
806     _v('CHECK 10', 'FAIL',

```

```

807         f'loss did not descend (drop={drop:.4f}); '
808         'gradient flow or optimizer issue')
809     return {'verdict': 'FAIL', 'drop': drop}
810
811
812 # CHECK 11 - HybridDiffusion distill prefix matching
813
814 def check_11() -> Dict:
815     _hdr('CHECK 11: HybridDiffusion distill prefix matching')
816     from models.diffusion_models import (
817         TDM, DiffusionPolicyModel, Diffuser, SDE, HybridDiffusion,
818     )
819     from config.base_config import GLOBAL_OBS_DIM
820
821     hd = HybridDiffusion(obs_dim=GLOBAL_OBS_DIM)
822     target_sd = hd.state_dict()
823     target_keys = set(target_sd.keys())
824
825     actual_submodules = {n for n, _ in hd.named_children()}
826     print(f' HybridDiffusion submodules: {sorted(actual_submodules)}')
827
828     run_src = Path('run.py').read_text()
829     m = re.search(r"sub_names\s*=\s*\{([^\}]*)\}", run_src, re.DOTALL)
830     declared_prefixes: Dict[str, str] = {}
831     if m:
832         for line in m.group(1).split(','):
833             mm = re.search(r"'([^\']+)'s*:\s*'([^\']+)'", line)
834             if mm:
835                 declared_prefixes[mm.group(1)] = mm.group(2)
836     print(f' run.py distill prefixes : {declared_prefixes}')
837
838     BASE = {
839         'TDM': TDM(obs_dim=GLOBAL_OBS_DIM),
840         'DiffusionPolicy': DiffusionPolicyModel(obs_dim=GLOBAL_OBS_DIM),

```

```

841         'Diffuser':          Diffuser(obs_dim=GLOBAL_OBS_DIM),
842         'SDE':              SDE(obs_dim=GLOBAL_OBS_DIM),
843     }
844
845     print()
846     print(f'  {"base":<18>{"prefix":<22>{"keys":<8>{"matched":<10>{"%
847         matched"}}')
848     total = 0
849     total_match = 0
850     bad_prefixes: List[Tuple[str, str]] = []
851     for base, model in BASE.items():
852         sd = model.state_dict()
853         pfx = declared_prefixes.get(base, '?')
854         n_match = 0
855         if pfx in actual_submodules:
856             for k, v in sd.items():
857                 tgt_key = f'{pfx}.{k}'
858                 if tgt_key in target_keys and target_sd[tgt_key].shape ==
859                     v.shape:
860                     n_match += 1
861             else:
862                 bad_prefixes.append((base, pfx))
863         n = len(sd)
864         pct = (100.0 * n_match / n) if n else 0.0
865         c = (_GREEN if n_match == n
866             else (_YEL if n_match > 0 else _RED))
867         valid_marker = '' if pfx in actual_submodules else ' [INVALID
868         PREFIX]'
869         print(f'  {c}{base:<18}{pfx:<22}{n:<8}{n_match:<10}'
870             f'{pct:>6.1f}%{valid_marker}{_END}')
871         total += n
872         total_match += n_match
873
874     pct_total = (100.0 * total_match / total) if total else 0.0

```



```

872     print(f'\n  TOTAL: {total_match}/{total} ({pct_total:.1f}%)')
873
874     if bad_prefixes:
875         print()
876         print(f'  {_RED}{_BOLD}BUG{_END}: run.py:_distill_hybrid_diff has
      ,
877             'invalid prefixes:')
878         for base, pfx in bad_prefixes:
879             print(f"      '{base}' → '{pfx}' (not a HybridDiffusion
submodule)")
880         print()
881         print('  HybridDiffusion submodule names are:')
882         for n in sorted(actual_submodules):
883             print(f'      {n}')
884         print()
885         print('  Suggested correction in run.py around line 354:')
886         print('      sub_names = {')
887         suggestions = {
888             'TDM':          'tdm',
889             'DiffusionPolicy': 'dp',
890             'Diffuser':      'diff',
891             'SDE':           'sde',
892         }
893         for base, sm in suggestions.items():
894             mark = '' if sm in actual_submodules else '  # CHECK THIS'
895             print(f"      '{base:<16}': '{sm}',{mark}")
896         print('    }')
897
898     if total_match >= total * 0.5 and not bad_prefixes:
899         _v('CHECK 11', 'PASS',
900             f'{total_match}/{total} matches and all prefixes valid')
901         return {'verdict': 'PASS', 'matched': total_match, 'total': total}
902     if bad_prefixes:
903         _v('CHECK 11', 'FAIL',

```

```

904         f'{len(bad_prefixes)} prefix(es) do not match HybridDiffusion '
905         'submodules --- only some base diffusion models will
warm-start')
906         return {'verdict': 'FAIL', 'bad_prefixes': bad_prefixes,
907                 'matched': total_match, 'total': total}
908     _v('CHECK 11', 'AMBIG',
909         f'{total_match}/{total} matches; partial warm-start')
910     return {'verdict': 'AMBIG', 'matched': total_match, 'total': total}
911
912
913 # =====
914     ===
915 # CHECKS 12--64 (K8--K64 from the spec verification matrix)
916 # =====
917     ===
918
919 # CHECK 12 - task status invariants post-reset (K8)
920
921 def check_12(scenario: str = 'warehouse') -> Dict:
922     _hdr(f'CHECK 12: all tasks have valid 4-state status ({scenario})')
923     from config.base_config import (
924         TASK_STATUS_UNAVAILABLE, TASK_STATUS_AVAILABLE,
925         TASK_STATUS_IN_PROGRESS, TASK_STATUS_COMPLETE,
926     )
927     valid = {TASK_STATUS_UNAVAILABLE, TASK_STATUS_AVAILABLE,
928             TASK_STATUS_IN_PROGRESS, TASK_STATUS_COMPLETE}
929     try:
930         env = ImprovedTaskAllocationEnv(scenario=scenario)
931         env.reset(seed=42)
932     except Exception as e:
933         _v('CHECK 12', 'FAIL', str(e))
934         return {'verdict': 'FAIL', 'error': str(e)}
935     bad = [t.get('status') for t in env.tasks if t.get('status') not in
valid]

```

```

934     from collections import Counter
935     for s, n in sorted(Counter(t.get('status') for t in
936         env.tasks).items()):
937         print(f'   {s:<20}: {n}')
938     if bad:
939         _v('CHECK 12', 'FAIL', f'{len(bad)} task(s) have unknown status:
940             {bad[:3]}')
941         return {'verdict': 'FAIL', 'bad': bad}
942     _v('CHECK 12', 'PASS', f'all {len(env.tasks)} tasks have valid
943         4-state statuses')
944     return {'verdict': 'PASS'}
945
946 # CHECK 13 - independent tasks start AVAILABLE (K9)
947
948 def check_13(scenario: str = 'warehouse') -> Dict:
949     _hdr(f'CHECK 13: tasks with no deps start as AVAILABLE ({scenario})')
950     from config.base_config import TASK_STATUS_AVAILABLE
951     try:
952         env = ImprovedTaskAllocationEnv(scenario=scenario)
953         env.reset(seed=42)
954     except Exception as e:
955         _v('CHECK 13', 'FAIL', str(e))
956         return {'verdict': 'FAIL', 'error': str(e)}
957     independent = [t for t in env.tasks if not t.get('dependencies')]
958     bad = [t for t in independent if t.get('status') !=
959         TASK_STATUS_AVAILABLE]
960     print(f'   independent tasks: {len(independent)}   wrong status:
961         {len(bad)}')
962     if bad:
963         _v('CHECK 13', 'FAIL', f'{len(bad)} independent task(s) not
964             AVAILABLE')
965         return {'verdict': 'FAIL', 'count': len(bad)}
966     if not independent:

```

```

962     _v('CHECK 13', 'AMBIG', 'no independent tasks in this episode')
963     return {'verdict': 'AMBIG'}
964     _v('CHECK 13', 'PASS',
965         f'all {len(independent)} independent tasks start as AVAILABLE')
966     return {'verdict': 'PASS'}
967
968
969 # CHECK 14 - dependent tasks start UNAVAILABLE (K10)
970
971 def check_14(scenario: str = 'warehouse') -> Dict:
972     _hdr(f'CHECK 14: tasks with deps start as UNAVAILABLE ({scenario})')
973     from config.base_config import TASK_STATUS_UNAVAILABLE
974     try:
975         env = ImprovedTaskAllocationEnv(scenario=scenario)
976         env.reset(seed=42)
977     except Exception as e:
978         _v('CHECK 14', 'FAIL', str(e))
979         return {'verdict': 'FAIL', 'error': str(e)}
980     dependent = [t for t in env.tasks if t.get('dependencies')]
981     bad = [t for t in dependent if t.get('status') !=
982           TASK_STATUS_UNAVAILABLE]
983     print(f' dependent tasks : {len(dependent)} wrong status:
984           {len(bad)}')
985     if bad:
986         _v('CHECK 14', 'FAIL',
987             f'{len(bad)} dependent task(s) not UNAVAILABLE at start')
988         return {'verdict': 'FAIL', 'count': len(bad)}
989     if not dependent:
990         _v('CHECK 14', 'AMBIG', 'no dependent tasks in this episode')
991         return {'verdict': 'AMBIG'}
992     _v('CHECK 14', 'PASS',
993         f'all {len(dependent)} dependent tasks start as UNAVAILABLE')
994     return {'verdict': 'PASS'}
995

```

```

994
995 # CHECK 15 - fleet size varies across resets (K11)
996
997 def check_15(scenario: str = 'warehouse', n_resets: int = 15) -> Dict:
998     sc = SCENARIO_CONFIGS[scenario]
999     _hdr(f'CHECK 15: num_robots varies
1000         [{sc.num_robots_min},{sc.num_robots_max}]'
1001         f' over {n_resets} resets ({scenario})')
1002     try:
1003         env = ImprovedTaskAllocationEnv(scenario=scenario)
1004         counts: List[int] = []
1005         for s in range(n_resets):
1006             env.reset(seed=s)
1007             counts.append(env.num_robots)
1008     except Exception as e:
1009         _v('CHECK 15', 'FAIL', str(e))
1010         return {'verdict': 'FAIL', 'error': str(e)}
1011     mn, mx = min(counts), max(counts)
1012     in_range = all(sc.num_robots_min <= c <= sc.num_robots_max for c in
1013                   counts)
1014     print(f'  observed range    : {mn} -- {mx}')
1015     print(f'  expected range    : {sc.num_robots_min} --
1016           {sc.num_robots_max}')
1017     print(f'  unique values      : {sorted(set(counts))}')
1018     if not in_range:
1019         _v('CHECK 15', 'FAIL', 'some count(s) outside [min, max]')
1020         return {'verdict': 'FAIL', 'min': mn, 'max': mx}
1021     if sc.num_robots_min < sc.num_robots_max and mx == mn:
1022         _v('CHECK 15', 'AMBIG', f'always got {mn} robots; sampling may be
1023           frozen')
1024         return {'verdict': 'AMBIG', 'min': mn, 'max': mx}
1025     _v('CHECK 15', 'PASS', f'fleet varies {mn}--{mx} within spec range')
1026     return {'verdict': 'PASS', 'min': mn, 'max': mx}

```

```

1024
1025 # CHECK 16 - dependency unlock (K12)
1026
1027 def check_16(scenario: str = 'warehouse') -> Dict:
1028     _hdr(f'CHECK 16: completing a dependency unlocks dependent tasks
1029         ({scenario})')
1030     from config.base_config import (TASK_STATUS_AVAILABLE,
1031                                     TASK_STATUS_UNAVAILABLE,
1032                                     TASK_STATUS_COMPLETE)
1033
1034     try:
1035         env = ImprovedTaskAllocationEnv(scenario=scenario)
1036         found = False
1037         for seed in range(30):
1038             env.reset(seed=seed)
1039             dep_tasks = [(i, t) for i, t in enumerate(env.tasks)
1040                           if t.get('dependencies') and
1041                           t.get('status') == TASK_STATUS_UNAVAILABLE]
1042
1043             if dep_tasks:
1044                 found = True
1045                 break
1046
1047     except Exception as e:
1048         _v('CHECK 16', 'FAIL', str(e))
1049         return {'verdict': 'FAIL', 'error': str(e)}
1050
1051     if not found:
1052         _v('CHECK 16', 'AMBIG', 'no UNAVAILABLE dependent tasks in 30
1053         seeds')
1054
1055         return {'verdict': 'AMBIG'}
1056
1057     dep_tid, dep_task = dep_tasks[0]
1058     prereqs = list(dep_task['dependencies'])
1059     print(f' target task {dep_tid} depends on {prereqs}')
1060
1061     # Mark all prereqs as COMPLETE
1062
1063     for pid in prereqs:
1064         env.tasks[pid]['status'] = TASK_STATUS_COMPLETE
1065         env.completed_tasks.add(pid)

```

```

1055     # Run the unlock loop (mirrors _complete_task logic)
1056     for t in env.tasks:
1057         if t.get('status') == TASK_STATUS_UNAVAILABLE:
1058             if all(d in env.completed_tasks for d in
1059                 t.get('dependencies', [])):
1060                 t['status'] = TASK_STATUS_AVAILABLE
1061                 new_status = env.tasks[dep_tid].get('status')
1062                 print(f'  status after unlock: {new_status}')
1063                 if new_status == TASK_STATUS_AVAILABLE:
1064                     _v('CHECK 16', 'PASS', f'task {dep_tid} unlocked to AVAILABLE')
1065                     return {'verdict': 'PASS'}
1066                 _v('CHECK 16', 'FAIL', f'task {dep_tid} still {new_status} after
1067                     prereqs complete')
1068                 return {'verdict': 'FAIL', 'status': new_status}
1069
1070
1071 # CHECK 17 - IN_PROGRESS tasks have exactly 1 assigned robot (K13)
1072
1073 def check_17(scenario: str = 'warehouse', n_steps: int = 100) -> Dict:
1074     _hdr(f'CHECK 17: IN_PROGRESS tasks assigned to exactly 1 robot
1075         ({scenario})')
1076     from config.base_config import TASK_STATUS_IN_PROGRESS
1077     try:
1078         env = ImprovedTaskAllocationEnv(scenario=scenario)
1079         env.reset(seed=42)
1080     except Exception as e:
1081         _v('CHECK 17', 'FAIL', str(e))
1082         return {'verdict': 'FAIL', 'error': str(e)}
1083     rng = np.random.RandomState(99)
1084     n_op = env.max_robots * env.max_tasks
1085     violations = 0
1086     for _ in range(n_steps):
1087         try:
1088             env.step(rng.randint(0, n_op + 1))

```

```

1086         except Exception:
1087             pass
1088         # task_assignments = {task_id: robot_id}; IN_PROGRESS task must
be a key
1089         for tid in range(env.num_tasks):
1090             if env.tasks[tid].get('status') == TASK_STATUS_IN_PROGRESS:
1091                 if tid not in env.task_assignments:
1092                     violations += 1
1093         print(f' steps run: {n_steps} violations: {violations}')
1094         if violations:
1095             _v('CHECK 17', 'FAIL', f'{violations} IN_PROGRESS task(s) had not
equal to 1 assignment')
1096             return {'verdict': 'FAIL', 'violations': violations}
1097         _v('CHECK 17', 'PASS', 'IN_PROGRESS exclusive-assignment invariant
holds')
1098         return {'verdict': 'PASS'}
1099
1100
1101 # CHECK 18 - COMPLETE tasks do not regress (K14)
1102
1103 def check_18(scenario: str = 'warehouse', n_steps: int = 200) -> Dict:
1104     _hdr(f'CHECK 18: COMPLETE tasks do not regress to other states
({scenario})')
1105     from config.base_config import TASK_STATUS_COMPLETE
1106     try:
1107         env = ImprovedTaskAllocationEnv(scenario=scenario)
1108         env.reset(seed=42)
1109     except Exception as e:
1110         _v('CHECK 18', 'FAIL', str(e))
1111         return {'verdict': 'FAIL', 'error': str(e)}
1112     rng = np.random.RandomState(77)
1113     n_op = env.max_robots * env.max_tasks
1114     regressions = 0
1115     for _ in range(n_steps):

```



```

1116         completed_before = {i for i, t in enumerate(env.tasks)
1117                               if t.get('status') == TASK_STATUS_COMPLETE}
1118     try:
1119         env.step(rng.randint(0, n_op + 1))
1120     except Exception:
1121         pass
1122     for tid in completed_before:
1123         if env.tasks[tid].get('status') != TASK_STATUS_COMPLETE:
1124             regressions += 1
1125     print(f'  steps: {n_steps}  regressions: {regressions}')
1126     if regressions:
1127         _v('CHECK 18', 'FAIL', f'{regressions} COMPLETE task(s)
1128         regressed')
1129         return {'verdict': 'FAIL', 'regressions': regressions}
1130     _v('CHECK 18', 'PASS', 'no COMPLETE task regressions observed')
1131     return {'verdict': 'PASS'}
1132
1133 # CHECK 19 - battery drains during movement (K18)
1134
1135 def check_19(scenario: str = 'warehouse') -> Dict:
1136     _hdr(f'CHECK 19: robot battery drains during movement ({scenario})')
1137     env = _build_env(scenario, num_robots=1, num_tasks=1, max_steps=500)
1138     env.reset(seed=42)
1139     cap = env.robots[0]['battery_capacity']
1140     env.robots[0]['current_battery'] = float(cap)
1141     half = float(min(env.environment['grid_size']) / 2.0)
1142     env.tasks[0].update({'location': [half * 0.8, 0.0]})
1143     env.robot_targets[0] = 0
1144     env.robots[0]['current_task'] = 0
1145     dt = env.dataset_config.DT
1146     for _ in range(80):
1147         if env.robots[0].get('operational_status') ==
1148             ROBOT_STATUS_INACTIVE:

```

```

1148         break
1149     try:
1150         env._update_robot_positions(dt)
1151     except Exception as e:
1152         _v('CHECK 19', 'FAIL', str(e))
1153         return {'verdict': 'FAIL', 'error': str(e)}
1154     final = env.robots[0]['current_battery']
1155     drained = float(cap) - final
1156     print(f'  battery_capacity : {cap:.2f}')
1157     print(f'  battery remaining: {final:.2f}')
1158     print(f'  drained           : {drained:.4f}')
1159     if drained > 0:
1160         _v('CHECK 19', 'PASS', f'battery drained by {drained:.4f}')
1161         return {'verdict': 'PASS', 'drained': drained}
1162     _v('CHECK 19', 'FAIL', 'battery did not drain during movement')
1163     return {'verdict': 'FAIL', 'drained': drained}
1164
1165
1166 # CHECK 20 - robot deactivates when battery exhausted (K19)
1167
1168 def check_20(scenario: str = 'warehouse') -> Dict:
1169     _hdr(f'CHECK 20: robot deactivates when battery exhausted
1170         ({scenario})')
1171     env = _build_env(scenario, num_robots=1, num_tasks=1, max_steps=200)
1172     env.reset(seed=42)
1173     env.robots[0]['current_battery'] = 0.0001 # Nearly empty
1174     half = float(min(env.environment['grid_size']) / 2.0)
1175     env.tasks[0].update({'location': [half * 0.8, 0.0]})
1176     env.robot_targets[0] = 0
1177     env.robots[0]['current_task'] = 0
1178     dt = env.dataset_config.DT
1179     deactivated_step = None
1180     for step in range(100):
1181         if env.robots[0].get('operational_status') ==

```

```

ROBOT_STATUS_INACTIVE:
1181         deactivated_step = step
1182         break
1183     try:
1184         env._update_robot_positions(dt)
1185     except Exception as e:
1186         _v('CHECK 20', 'FAIL', str(e))
1187         return {'verdict': 'FAIL', 'error': str(e)}
1188     print(f'  deactivated at step: {deactivated_step}')
1189     if deactivated_step is not None:
1190         _v('CHECK 20', 'PASS', f'robot deactivated at step
{deactivated_step}')
1191         return {'verdict': 'PASS', 'step': deactivated_step}
1192     _v('CHECK 20', 'FAIL', 'robot not deactivated after 100 steps with
near-zero battery')
1193     return {'verdict': 'FAIL'}
1194
1195
1196 # CHECK 21 - deactivated robot has status 'inactive' (K20)
1197
1198 def check_21(scenario: str = 'warehouse') -> Dict:
1199     _hdr(f'CHECK 21: deactivated robot has
operational_status=\`inactive\` ({scenario})')
1200     env = _build_env(scenario, num_robots=1, num_tasks=1, max_steps=100)
1201     env.reset(seed=42)
1202     env.robots[0]['current_battery'] = 0.0 # Force deactivation
1203     half = float(min(env.environment['grid_size']) / 2.0)
1204     env.tasks[0].update({'location': [half * 0.5, 0.0]})
1205     env.robot_targets[0] = 0
1206     env.robots[0]['current_task'] = 0
1207     dt = env.dataset_config.DT
1208     try:
1209         env._update_robot_positions(dt)
1210     except Exception as e:

```

```

1211         _v('CHECK 21', 'FAIL', str(e))
1212         return {'verdict': 'FAIL', 'error': str(e)}
1213     status = env.robots[0].get('operational_status')
1214     battery = env.robots[0]['current_battery']
1215     print(f' battery after step: {battery:.4f}')
1216     print(f' operational_status: {status!r}')
1217     if status == ROBOT_STATUS_INACTIVE:
1218         _v('CHECK 21', 'PASS', f"deactivated robot status =
1219         {ROBOT_STATUS_INACTIVE!r}")
1220         return {'verdict': 'PASS'}
1221     _v('CHECK 21', 'FAIL', f"expected {ROBOT_STATUS_INACTIVE!r}, got
1222     {status!r}")
1223     return {'verdict': 'FAIL', 'status': status}
1224
1225 # CHECK 22 - hex interior cell returns 6 neighbours (K21)
1226
1227 def check_22() -> Dict:
1228     _hdr('CHECK 22: hex get_neighbours --- interior cell returns 6
1229     neighbours')
1230     from utils.hex_utils import get_neighbours
1231     W, H = 20, 20
1232     interior = 10 * W + 10 # col=10, row=10
1233     nbrs = get_neighbours(interior, W, H)
1234     print(f' grid : {W}×{H}')
1235     print(f' interior cell : {interior} (col=10, row=10)')
1236     print(f' neighbour count : {len(nbrs)}')
1237     if len(nbrs) == 6:
1238         _v('CHECK 22', 'PASS', 'interior hex cell has exactly 6
1239         neighbours')
1240         return {'verdict': 'PASS'}
1241     _v('CHECK 22', 'FAIL', f'expected 6, got {len(nbrs)}')
1242     return {'verdict': 'FAIL', 'count': len(nbrs)}

```

```

1241
1242 # CHECK 23 - hex corner cell returns < 6 neighbours (K22)
1243
1244 def check_23() -> Dict:
1245     _hdr('CHECK 23: hex get_neighbours --- corner cell returns < 6
1246         neighbours')
1247     from utils.hex_utils import get_neighbours
1248     W, H = 20, 20
1249     corner = 0 # col=0, row=0
1250     nbrs = get_neighbours(corner, W, H)
1251     print(f' corner cell (0,0) neighbour count: {len(nbrs)}')
1252     if len(nbrs) < 6:
1253         _v('CHECK 23', 'PASS', f'corner cell has {len(nbrs)} < 6
1254             neighbours')
1255         return {'verdict': 'PASS', 'count': len(nbrs)}
1256     _v('CHECK 23', 'FAIL', f'expected < 6, got {len(nbrs)}')
1257     return {'verdict': 'FAIL', 'count': len(nbrs)}
1258
1259 # CHECK 24 - hex_distance adjacent cells = 1 (K23)
1260
1261 def check_24() -> Dict:
1262     _hdr('CHECK 24: hex_distance between adjacent cells = 1')
1263     from utils.hex_utils import get_neighbours, hex_distance
1264     W = 20
1265     center = 10 * W + 10
1266     nbrs = get_neighbours(center, W, W)
1267     if not nbrs:
1268         _v('CHECK 24', 'FAIL', 'no neighbours returned')
1269         return {'verdict': 'FAIL'}
1270     dists = [hex_distance(center, n, W) for n in nbrs]
1271     print(f' distances to neighbours: {dists}')
1272     if all(d == 1 for d in dists):
1273         _v('CHECK 24', 'PASS', 'all neighbours have hex_distance = 1')

```

```

1273         return {'verdict': 'PASS'}
1274     bad = [d for d in dists if d != 1]
1275     _v('CHECK 24', 'FAIL', f'non-unit distances: {bad}')
1276     return {'verdict': 'FAIL', 'bad': bad}
1277
1278
1279 # CHECK 25 - hex_distance self = 0 (K24)
1280
1281 def check_25() -> Dict:
1282     _hdr('CHECK 25: hex_distance(cell, cell) = 0')
1283     from utils.hex_utils import hex_distance
1284     W = 20
1285     for cid in [0, 5, 10 * W + 10, 19 * W + 19]:
1286         d = hex_distance(cid, cid, W)
1287         if d != 0:
1288             _v('CHECK 25', 'FAIL', f'hex_distance({cid},{cid}) = {d},
expected 0')
1289             return {'verdict': 'FAIL', 'cell': cid, 'distance': d}
1290     _v('CHECK 25', 'PASS', 'hex_distance(c, c) = 0 for all tested cells')
1291     return {'verdict': 'PASS'}
1292
1293
1294 # CHECK 26 - hex_ring cells have correct distance (K25)
1295
1296 def check_26() -> Dict:
1297     _hdr('CHECK 26: hex_ring cells all have distance == radius from
center')
1298     from utils.hex_utils import hex_ring, hex_distance
1299     W, H = 20, 20
1300     center = 10 * W + 10
1301     failures = []
1302     for r in range(1, 5):
1303         ring = hex_ring(center, r, W, H)
1304         wrong = [c for c in ring if hex_distance(center, c, W) != r]

```

```

1305         print(f' ring r={r}: {len(ring)} cells, wrong distance:
1306               {len(wrong)}')
1307         failures.extend(wrong)
1308     if failures:
1309         _v('CHECK 26', 'FAIL', f'{len(failures)} cells with wrong ring
1310               distance')
1311         return {'verdict': 'FAIL'}
1312     _v('CHECK 26', 'PASS', 'all hex_ring cells have correct distance from
1313           center')
1314     return {'verdict': 'PASS'}
1315
1316 # CHECK 27 - hex_ring(c, 1) matches get_neighbours(c) (K26)
1317
1318 def check_27() -> Dict:
1319     _hdr('CHECK 27: hex_ring(c, 1) == sorted(get_neighbours(c))')
1320     from utils.hex_utils import hex_ring, get_neighbours
1321     W, H = 20, 20
1322     center = 10 * W + 10
1323     ring1 = sorted(hex_ring(center, 1, W, H))
1324     nbrs = sorted(get_neighbours(center, W, H))
1325     print(f' hex_ring(c,1) : {ring1}')
1326     print(f' get_neighbours(c) : {nbrs}')
1327     if ring1 == nbrs:
1328         _v('CHECK 27', 'PASS', 'hex_ring(c,1) == get_neighbours(c)')
1329         return {'verdict': 'PASS'}
1330     _v('CHECK 27', 'FAIL', 'mismatch between hex_ring(c,1) and
1331           get_neighbours(c)')
1332     return {'verdict': 'FAIL', 'ring1': ring1, 'nbrs': nbrs}
1333
1334 # CHECK 28 - cells_within_range
1335
1336 def check_28() -> Dict:

```

```

1335     _hdr('CHECK 28: cells_within_range')
1336     from utils.hex_utils import cells_within_range, get_neighbours
1337     W, H = 20, 20
1338     center = 10 * W + 10
1339     within1 = sorted(cells_within_range(center, 1, W, H))
1340     expected = sorted({center} | set(get_neighbours(center, W, H)))
1341     print(f'  cells_within_range(c,1) count: {len(within1)}')
1342     print(f'  expected (c + neighbours)      : {len(expected)}')
1343     if within1 == expected:
1344         _v('CHECK 28', 'PASS')
1345         return {'verdict': 'PASS'}
1346     _v('CHECK 28', 'FAIL', 'mismatch')
1347     return {'verdict': 'FAIL'}
1348
1349
1350 # CHECK 29 - cells_within_range(c, 0) = [c] (K28)
1351
1352 def check_29() -> Dict:
1353     _hdr('CHECK 29: cells_within_range(c, 0) = [center]')
1354     from utils.hex_utils import cells_within_range
1355     W = 20
1356     for center in [0, 5, 10 * W + 10]:
1357         result = cells_within_range(center, 0, W, W)
1358         if result != [center]:
1359             _v('CHECK 29', 'FAIL',
1360                 f'cells_within_range({center}, 0) = {result}, expected
1361                 [{center}]')
1362             return {'verdict': 'FAIL'}
1363         _v('CHECK 29', 'PASS', 'cells_within_range(c, 0) = [c] for all tested
1364             cells')
1365         return {'verdict': 'PASS'}
1366
1367 # CHECK 30 - edge cell has fewer than 6 hex neighbours (K29)

```



```

1367
1368 def check_30() -> Dict:
1369     _hdr('CHECK 30: edge cell has fewer than 6 hex neighbours')
1370     from utils.hex_utils import get_neighbours
1371     W, H = 20, 20
1372     edge = W - 1 # col=19, row=0 (top-right corner)
1373     nbrs = get_neighbours(edge, W, H)
1374     print(f' edge cell {edge} (col=19, row=0) neighbour count:
1375           {len(nbrs)}')
1375     if len(nbrs) < 6:
1376         _v('CHECK 30', 'PASS', f'edge cell has {len(nbrs)} < 6
1377           neighbours')
1378         return {'verdict': 'PASS', 'count': len(nbrs)}
1379     _v('CHECK 30', 'FAIL', f'expected < 6, got {len(nbrs)}')
1380     return {'verdict': 'FAIL', 'count': len(nbrs)}
1381
1382 # CHECK 31 - action mask shape (K34)
1383
1384 def check_31(scenario: str = 'warehouse') -> Dict:
1385     _hdr(f'CHECK 31: build_action_mask returns correct shape
1386           ({scenario})')
1387     from utils.action_masking import build_action_mask
1388     try:
1389         env = ImprovedTaskAllocationEnv(scenario=scenario)
1390         env.reset(seed=42)
1391     except Exception as e:
1392         _v('CHECK 31', 'FAIL', str(e))
1393         return {'verdict': 'FAIL', 'error': str(e)}
1394     num_r = env.num_robots
1395     max_t = env.max_tasks
1396     mask = build_action_mask(env.tasks, num_r, max_t)
1397     expected = (num_r * max_t + 1,)
1398     print(f' num_robots={num_r} max_tasks={max_t}')

```

```

1398     print(f'   mask shape   : {tuple(mask.shape)}')
1399     print(f'   expected     : {expected}')
1400     if tuple(mask.shape) == expected:
1401         _v('CHECK 31', 'PASS', f'mask shape {expected} correct')
1402         return {'verdict': 'PASS'}
1403     _v('CHECK 31', 'FAIL', 'shape mismatch')
1404     return {'verdict': 'FAIL', 'shape': tuple(mask.shape)}
1405
1406
1407 # CHECK 32 - UNAVAILABLE tasks masked (K35)
1408
1409 def check_32() -> Dict:
1410     _hdr('CHECK 32: build_action_mask --- UNAVAILABLE tasks are invalid')
1411     from utils.action_masking import build_action_mask
1412     from config.base_config import TASK_STATUS_UNAVAILABLE,
1413     TASK_STATUS_AVAILABLE
1414     tasks = [
1415         {'status': TASK_STATUS_UNAVAILABLE, 'dependencies': [1]},
1416         {'status': TASK_STATUS_AVAILABLE, 'dependencies': []},
1417         {'status': TASK_STATUS_UNAVAILABLE, 'dependencies': [1]},
1418     ]
1419     mask = build_action_mask(tasks, num_robots=1, max_tasks=3)
1420     t0_bad = bool(mask[0].item()) # UNAVAILABLE → should be True
1421     (invalid)
1422     t1_ok = not bool(mask[1].item()) # AVAILABLE → should be False
1423     (valid)
1424     t2_bad = bool(mask[2].item()) # UNAVAILABLE → should be True
1425     print(f'   task 0 UNAVAILABLE masked: {t0_bad}')
1426     print(f'   task 1 AVAILABLE   valid : {t1_ok}')
1427     print(f'   task 2 UNAVAILABLE masked: {t2_bad}')
1428     if t0_bad and t1_ok and t2_bad:
1429         _v('CHECK 32', 'PASS', 'UNAVAILABLE tasks correctly masked')
1430         return {'verdict': 'PASS'}
1431     _v('CHECK 32', 'FAIL', 'masking incorrect for UNAVAILABLE tasks')

```

```

1429     return {'verdict': 'FAIL'}
1430
1431
1432 # CHECK 33 - COMPLETE tasks masked (K36)
1433
1434 def check_33() -> Dict:
1435     _hdr('CHECK 33: build_action_mask --- COMPLETE tasks are invalid')
1436     from utils.action_masking import build_action_mask
1437     from config.base_config import TASK_STATUS_COMPLETE,
1438         TASK_STATUS_AVAILABLE
1439     tasks = [
1440         {'status': TASK_STATUS_COMPLETE, 'dependencies': []},
1441         {'status': TASK_STATUS_AVAILABLE, 'dependencies': []},
1442     ]
1443     mask = build_action_mask(tasks, num_robots=1, max_tasks=2)
1444     t0_masked = bool(mask[0].item())
1445     t1_valid = not bool(mask[1].item())
1446     print(f' task 0 (COMPLETE) masked: {t0_masked}')
1447     print(f' task 1 (AVAILABLE) valid: {t1_valid}')
1448     if t0_masked and t1_valid:
1449         _v('CHECK 33', 'PASS', 'COMPLETE tasks correctly masked')
1450         return {'verdict': 'PASS'}
1451     _v('CHECK 33', 'FAIL', 'COMPLETE not masked or AVAILABLE wrongly
1452         masked')
1453     return {'verdict': 'FAIL'}
1454
1455
1456 # CHECK 34 - PCGrad conflicting gradients (K37)
1457
1458 def check_34() -> Dict:
1459     _hdr('CHECK 34: PCGrad handles conflicting gradients without
1460         explosion')
1461     from utils.pcgrad import pcgrad_step
1462     p = torch.nn.Parameter(torch.zeros(4))

```

```

1460     # Two perfectly opposing gradient directions
1461     loss_a = p[0] + p[1]    # grad ≈ [1, 1, 0, 0]
1462     loss_b = -p[0] - p[1]   # grad ≈ [-1, -1, 0, 0] (opposing)
1463     try:
1464         grads = pcgrad_step([loss_a, loss_b], [p])
1465     except Exception as e:
1466         _v('CHECK 34', 'FAIL', f'{type(e).__name__}: {e}')
1467         return {'verdict': 'FAIL', 'error': str(e)}
1468     g = grads[0]
1469     if g is None:
1470         _v('CHECK 34', 'FAIL', 'PCGrad returned None gradient')
1471         return {'verdict': 'FAIL'}
1472     norm = float(g.norm())
1473     finite = not bool((g != g).any()) and norm < 1e6
1474     print(f'  gradient result   : {g.tolist()}')
1475     print(f'  L2 norm           : {norm:.6f}')
1476     print(f'  finite and bounded: {finite}')
1477     if finite:
1478         _v('CHECK 34', 'PASS', f'PCGrad handled conflicting grads;
norm={norm:.4f}')
1479         return {'verdict': 'PASS', 'norm': norm}
1480     _v('CHECK 34', 'FAIL', f'gradient exploded or NaN (norm={norm})')
1481     return {'verdict': 'FAIL', 'norm': norm}
1482
1483
1484 # CHECK 35 - PCGrad aligned gradients (K38)
1485
1486 def check_35() -> Dict:
1487     _hdr('CHECK 35: PCGrad with aligned gradients equals plain sum')
1488     from utils.pcgrad import pcgrad_step
1489     p = torch.nn.Parameter(torch.zeros(2))
1490     tgt_a = torch.tensor([1.0, 0.0])
1491     tgt_b = torch.tensor([0.5, 0.0])
1492     loss_a = (p - tgt_a).pow(2).sum()

```

```

1493     loss_b = (p - tgt_b).pow(2).sum()
1494     try:
1495         grads = pcgrad_step([loss_a, loss_b], [p])
1496     except Exception as e:
1497         _v('CHECK 35', 'FAIL', f'{type(e).__name__}: {e}')
1498         return {'verdict': 'FAIL', 'error': str(e)}
1499     g = grads[0]
1500     # Plain sum for comparison
1501     p2 = torch.nn.Parameter(torch.zeros(2))
1502     l2a = (p2 - tgt_a).pow(2).sum()
1503     l2b = (p2 - tgt_b).pow(2).sum()
1504     g_sum = torch.autograd.grad(l2a + l2b, [p2])[0]
1505     print(f' plain sum grad : {g_sum.tolist()}')
1506     print(f' PCGrad result : {g.tolist() if g is not None else None}')
1507     if g is None:
1508         _v('CHECK 35', 'FAIL', 'PCGrad returned None')
1509         return {'verdict': 'FAIL'}
1510     close = torch.allclose(g, g_sum, atol=1e-4)
1511     if close:
1512         _v('CHECK 35', 'PASS', 'aligned PCGrad == plain sum')
1513         return {'verdict': 'PASS'}
1514     _v('CHECK 35', 'AMBIG', f'not exactly equal --- diff norm {(g -
1515         g_sum).norm():.4f}')
1516     return {'verdict': 'AMBIG'}
1517
1518 # CHECK 36 - QMIX forward with N=2 robots (K39)
1519
1520 def check_36() -> Dict:
1521     _hdr('CHECK 36: QMIX forward pass with N=2 robots')
1522     from models.rl_agents import QMIX
1523     from config.base_config import LOCAL_OBS_DIM
1524     qmix = QMIX()
1525     local_obs = torch.randn(2, LOCAL_OBS_DIM)

```

```

1526     with torch.no_grad():
1527         _, task_id = qmix(local_obs)
1528         # task_id may be (N,) batch tensor --- check all elements
1529         tids = task_id.tolist() if task_id.dim() > 0 else [task_id.item()]
1530         print(f' N=2 task_ids={tids}')
1531         bad = [t for t in tids if not (0 <= t < QMIX.MAX_TASKS)]
1532         if not bad:
1533             _v('CHECK 36', 'PASS', f'N=2 task_ids in [0, {QMIX.MAX_TASKS}]')
1534             return {'verdict': 'PASS'}
1535         _v('CHECK 36', 'FAIL', f'task_ids out of range: {bad}')
1536         return {'verdict': 'FAIL', 'bad': bad}
1537
1538
1539 # CHECK 37 - QMIX forward with N=20 robots (K40)
1540
1541 def check_37() -> Dict:
1542     _hdr('CHECK 37: QMIX forward pass with N=20 robots')
1543     from models.rl_agents import QMIX
1544     from config.base_config import LOCAL_OBS_DIM
1545     qmix = QMIX()
1546     local_obs = torch.randn(20, LOCAL_OBS_DIM)
1547     with torch.no_grad():
1548         _, task_id = qmix(local_obs)
1549         tids = task_id.tolist() if task_id.dim() > 0 else [task_id.item()]
1550         print(f' N=20 task_ids sample: {tids[:5]}...')
1551         bad = [t for t in tids if not (0 <= t < QMIX.MAX_TASKS)]
1552         if not bad:
1553             _v('CHECK 37', 'PASS', f'N=20 task_ids in [0, {QMIX.MAX_TASKS}]')
1554             return {'verdict': 'PASS'}
1555         _v('CHECK 37', 'FAIL', f'task_ids out of range: {bad}')
1556         return {'verdict': 'FAIL', 'bad': bad}
1557
1558
1559 # CHECK 38 - ReplayBuffer push / sample / overflow (K41)

```

```

1560
1561 def check_38() -> Dict:
1562     _hdr('CHECK 38: ReplayBuffer push, sample, len, and overflow')
1563     from training.rl_trainer import ReplayBuffer
1564     obs_dim = 10
1565     buf = ReplayBuffer(capacity=100)
1566     if len(buf) != 0:
1567         _v('CHECK 38', 'FAIL', f'buffer should start empty,
len={len(buf)}')
1568         return {'verdict': 'FAIL'}
1569     for i in range(50):
1570         obs = np.random.randn(obs_dim).astype(np.float32)
1571         nobs = np.random.randn(obs_dim).astype(np.float32)
1572         buf.add(obs, i % 5, float(i), nobs, float(i == 49))
1573     if len(buf) != 50:
1574         _v('CHECK 38', 'FAIL', f'expected 50, got {len(buf)}')
1575         return {'verdict': 'FAIL'}
1576     try:
1577         obs_b, act_b, rew_b, nobs_b, done_b = buf.sample(batch_size=16)
1578     except Exception as e:
1579         _v('CHECK 38', 'FAIL', f'sample crashed: {e}')
1580         return {'verdict': 'FAIL', 'error': str(e)}
1581     ok = (obs_b.shape == (16, obs_dim) and act_b.shape == (16,))
1582     for i in range(60):
1583         obs = np.random.randn(obs_dim).astype(np.float32)
1584         nobs = np.random.randn(obs_dim).astype(np.float32)
1585         buf.add(obs, 0, 1.0, nobs, 0.0)
1586     print(f' after 50 pushes: len={50} sample shapes OK: {ok}')
1587     print(f' after overflow : len={len(buf)} (capacity={buf.capacity})')
1588     if len(buf) != 100:
1589         _v('CHECK 38', 'FAIL', f'overflow: expected 100, got {len(buf)}')
1590         return {'verdict': 'FAIL'}
1591     if ok:
1592         _v('CHECK 38', 'PASS', 'ReplayBuffer push/sample/overflow all

```

```

work')
1593     return {'verdict': 'PASS'}
1594     _v('CHECK 38', 'FAIL', 'sample shape mismatch')
1595     return {'verdict': 'FAIL'}
1596
1597
1598 # CHECK 39 - TDM.sample returns (n_samples, n_eval, 2) (K46)
1599
1600 def check_39() -> Dict:
1601     _hdr('CHECK 39: TDM.sample returns (n_samples, n_eval, 2)')
1602     from models.diffusion_models import TDM
1603     from config.base_config import GLOBAL_OBS_DIM
1604     n_eval = 20
1605     torch.manual_seed(0)
1606     model = TDM(obs_dim=GLOBAL_OBS_DIM).eval()
1607     obs      = torch.randn(1, GLOBAL_OBS_DIM)
1608     start_pos = torch.zeros(1, 2)
1609     goal_pos  = torch.ones(1, 2) * 0.5
1610     try:
1611         with torch.no_grad():
1612             samples = model.sample(obs, start_pos, goal_pos, n_samples=3)
1613     except Exception as e:
1614         _v('CHECK 39', 'FAIL', f'{type(e).__name__}: {e}')
1615         return {'verdict': 'FAIL', 'error': str(e)}
1616     expected = (3, n_eval, 2)
1617     print(f'  TDM.sample output shape: {tuple(samples.shape)}')
1618     if tuple(samples.shape) == expected:
1619         _v('CHECK 39', 'PASS', f'TDM.sample → {expected}')
1620         return {'verdict': 'PASS'}
1621     _v('CHECK 39', 'FAIL', f'expected {expected}, got
        {tuple(samples.shape)}')
1622     return {'verdict': 'FAIL', 'shape': tuple(samples.shape)}
1623
1624

```



```

1625 # CHECK 40 - DiffusionPolicyModel forward (K47)
1626
1627 def check_40() -> Dict:
1628     _hdr('CHECK 40: DiffusionPolicyModel forward runs without error')
1629     from models.diffusion_models import DiffusionPolicyModel
1630     from config.base_config import GLOBAL_OBS_DIM
1631     torch.manual_seed(1)
1632     model = DiffusionPolicyModel(obs_dim=GLOBAL_OBS_DIM).eval()
1633     obs      = torch.randn(1, GLOBAL_OBS_DIM)
1634     task_locs = torch.randn(3, 2)
1635     other     = torch.zeros(0, 2)
1636     try:
1637         with torch.no_grad():
1638             out = model(obs, task_locs, other)
1639     except Exception as e:
1640         _v('CHECK 40', 'FAIL', f'{type(e).__name__}: {e}')
1641         return {'verdict': 'FAIL', 'error': str(e)}
1642     print(f' DiffusionPolicy output type: {type(out).__name__}')
1643     _v('CHECK 40', 'PASS', 'DiffusionPolicyModel forward completed
        without error')
1644     return {'verdict': 'PASS'}
1645
1646
1647 # CHECK 41 - TrajectoryDataset.augment produces 5× data (K48)
1648
1649 def check_41() -> Dict:
1650     _hdr('CHECK 41: TrajectoryDataset.augment produces 5× data (1
        original + 4 variants)')
1651     from training.diffusion_trainer import TrajectoryDataset
1652     base = [{'obs':      [0.0] * 10,
1653              'start_pos': [0.0, 0.0],
1654              'goal_pos':  [1.0, 1.0],
1655              'waypoints': [[0.0, 0.0], [0.5, 0.5], [1.0, 1.0]]}]
1656     aug = TrajectoryDataset.augment(base)

```

```

1657     print(f'   original samples: {len(base)}')
1658     print(f'   after augment   : {len(aug)}')
1659     if len(aug) == 5:
1660         _v('CHECK 41', 'PASS', f'augment produces 5× data')
1661         return {'verdict': 'PASS', 'count': len(aug)}
1662     _v('CHECK 41', 'FAIL', f'expected 5, got {len(aug)}')
1663     return {'verdict': 'FAIL', 'count': len(aug)}
1664
1665
1666 # CHECK 42 - augment flip-x negates x-coordinates (K49)
1667
1668 def check_42() -> Dict:
1669     _hdr('CHECK 42: augment flip-x negates x-coordinates')
1670     from training.diffusion_trainer import TrajectoryDataset
1671     wps = [[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]
1672     base = [{'obs': [0.0], 'start_pos': [1.0, 0.5],
1673             'goal_pos': [2.0, 3.0], 'waypoints': wps}]
1674     aug = TrajectoryDataset.augment(base)
1675     flip_x = aug[1] # index 1 = flip-x variant
1676     orig_wp = np.array(wps)
1677     flip_wp = np.array(flip_x['waypoints'])
1678     x_ok = np.allclose(flip_wp[:, 0], -orig_wp[:, 0])
1679     y_ok = np.allclose(flip_wp[:, 1], orig_wp[:, 1])
1680     sx_ok = flip_x['start_pos'][0] == -base[0]['start_pos'][0]
1681     print(f'   orig x: {orig_wp[:, 0].tolist()}')
1682     print(f'   flip x: {flip_wp[:, 0].tolist()}')
1683     print(f'   x negated={x_ok}   y unchanged={y_ok}   start_x ok={sx_ok}')
1684     if x_ok and y_ok and sx_ok:
1685         _v('CHECK 42', 'PASS', 'flip-x correctly negates x-coordinates')
1686         return {'verdict': 'PASS'}
1687     _v('CHECK 42', 'FAIL', f'flip-x incorrect: x_ok={x_ok}, y_ok={y_ok}')
1688     return {'verdict': 'FAIL'}
1689
1690

```

```

1691 # CHECK 43 - SDE forward no NaN (K50)
1692
1693 def check_43() -> Dict:
1694     _hdr('CHECK 43: SDE forward pass produces no NaN values')
1695     from models.diffusion_models import SDE
1696     from config.base_config import GLOBAL_OBS_DIM
1697     torch.manual_seed(2)
1698     model = SDE(obs_dim=GLOBAL_OBS_DIM).eval()
1699     obs      = torch.randn(1, GLOBAL_OBS_DIM)
1700     task_locs = torch.randn(3, 2)
1701     other     = torch.zeros(0, 2)
1702     try:
1703         with torch.no_grad():
1704             out = model(obs, task_locs, other)
1705     except Exception as e:
1706         _v('CHECK 43', 'FAIL', f'{type(e).__name__}: {e}')
1707         return {'verdict': 'FAIL', 'error': str(e)}
1708     tensors = ([t for t in out if torch.is_tensor(t)]
1709                if isinstance(out, tuple) else
1710                ([out] if torch.is_tensor(out) else []))
1711     has_nan = any(t.isnan().any() for t in tensors)
1712     has_inf = any(t.isinf().any() for t in tensors)
1713     print(f' has NaN: {has_nan} has Inf: {has_inf}')
1714     if not has_nan and not has_inf:
1715         _v('CHECK 43', 'PASS', 'SDE forward produces no NaN/Inf')
1716         return {'verdict': 'PASS'}
1717     _v('CHECK 43', 'FAIL', f'NaN={has_nan} or Inf={has_inf} in SDE
1718     output')
1719     return {'verdict': 'FAIL', 'has_nan': has_nan}
1720
1721 # CHECK 44 - HybridDiffusion returns (robot_id, task_id) (K51)
1722
1723 def check_44() -> Dict:

```

```

1724     _hdr('CHECK 44: HybridDiffusion forward returns (robot_id, task_id)
        tuple')
1725     from models.diffusion_models import HybridDiffusion
1726     from config.base_config import GLOBAL_OBS_DIM
1727     torch.manual_seed(3)
1728     model = HybridDiffusion(obs_dim=GLOBAL_OBS_DIM).eval()
1729     obs      = torch.randn(1, GLOBAL_OBS_DIM)
1730     task_locs = torch.randn(2, 2)
1731     other     = torch.zeros(0, 2)
1732     try:
1733         with torch.no_grad():
1734             out = model(obs, task_locs, other)
1735     except Exception as e:
1736         _v('CHECK 44', 'FAIL', f'{type(e).__name__}: {e}')
1737         return {'verdict': 'FAIL', 'error': str(e)}
1738     if isinstance(out, tuple) and len(out) == 2:
1739         rid, tid = out
1740         print(f'  robot_id={int(rid.item())}, task_id={int(tid.item())}')
1741         _v('CHECK 44', 'PASS', 'HybridDiffusion returns (robot_id,
        task_id)')
1742         return {'verdict': 'PASS'}
1743     _v('CHECK 44', 'FAIL', f'unexpected output: {type(out).__name__}')
1744     return {'verdict': 'FAIL'}
1745
1746
1747 # CHECK 45 - SCENARIO_TASK_TYPES has 3 keys (K52)
1748
1749 def check_45() -> Dict:
1750     _hdr('CHECK 45: SCENARIO_TASK_TYPES contains all 3 scenario keys')
1751     from config.base_config import SCENARIO_TASK_TYPES
1752     expected = {'warehouse', 'search_and_rescue', 'maritime_security'}
1753     actual   = set(SCENARIO_TASK_TYPES.keys())
1754     print(f'  expected: {sorted(expected)}')
1755     print(f'  actual   : {sorted(actual)}')

```

```

1756     if actual == expected:
1757         _v('CHECK 45', 'PASS', '3 scenario keys present')
1758         return {'verdict': 'PASS'}
1759     missing = expected - actual
1760     extra = actual - expected
1761     _v('CHECK 45', 'FAIL', f'missing={missing}, extra={extra}')
1762     return {'verdict': 'FAIL', 'missing': list(missing)}
1763
1764
1765 # CHECK 46 - task pool sizes (K53)
1766
1767 def check_46() -> Dict:
1768     _hdr('CHECK 46: task pool sizes --- warehouse=12, SAR=10,
1769         maritime=10')
1770     from config.base_config import (WAREHOUSE_TASK_TYPES, SAR_TASK_TYPES,
1771                                     MARITIME_TASK_TYPES)
1772     expected = [
1773         ('warehouse', WAREHOUSE_TASK_TYPES, 12),
1774         ('search_and_rescue', SAR_TASK_TYPES, 10),
1775         ('maritime_security', MARITIME_TASK_TYPES, 10),
1776     ]
1777     failures = []
1778     for name, pool, size in expected:
1779         n = len(pool)
1780         ok = (n == size)
1781         c = _GREEN if ok else _RED
1782         print(f' {c}{name:<22}: {n} (expected {size}){_END}')
1783         if not ok:
1784             failures.append(f'{name}: {n} not equal to {size}')
1785     if failures:
1786         _v('CHECK 46', 'FAIL', '; '.join(failures))
1787         return {'verdict': 'FAIL', 'failures': failures}
1788     _v('CHECK 46', 'PASS', 'all pool sizes correct')
1789     return {'verdict': 'PASS'}

```

```

1789
1790
1791 # CHECK 47 - no circular dependencies in task type DAGs (K54)
1792
1793 def check_47() -> Dict:
1794     _hdr('CHECK 47: no circular dependencies in task type DAGs')
1795     from config.base_config import SCENARIO_TASK_TYPES
1796
1797     def _has_cycle(pool):
1798         id_set = {t['id'] for t in pool}
1799         adj = {t['id']: [d for d in t.get('depends_on', []) if d in
1800             id_set]
1801             for t in pool}
1802         WHITE, GREY, BLACK = 0, 1, 2
1803         colour = {tid: WHITE for tid in id_set}
1804
1805         def dfs(node, path):
1806             colour[node] = GREY
1807             for nb in adj.get(node, []):
1808                 if colour[nb] == GREY:
1809                     return path + [nb]
1810                 if colour[nb] == WHITE:
1811                     res = dfs(nb, path + [nb])
1812                     if res:
1813                         return res
1814             colour[node] = BLACK
1815             return None
1816
1817         for node in id_set:
1818             if colour[node] == WHITE:
1819                 cycle = dfs(node, [node])
1820                 if cycle:
1821                     return cycle
1822         return None

```

```

1822
1823     failures = []
1824     for scenario, pool in SCENARIO_TASK_TYPES.items():
1825         cycle = _has_cycle(pool)
1826         ok = cycle is None
1827         c = _GREEN if ok else _RED
1828         print(f' {c}{scenario:<22}: {"no cycle" if ok else f"CYCLE"}{cycle}"{_END}')
1829         if not ok:
1830             failures.append(f'{scenario}: {cycle}')
1831     if failures:
1832         _v('CHECK 47', 'FAIL', f'cycles found: {failures}')
1833         return {'verdict': 'FAIL', 'cycles': failures}
1834     _v('CHECK 47', 'PASS', 'no circular dependencies in any task type DAG')
1835     return {'verdict': 'PASS'}
1836
1837
1838 # CHECK 48 - depends_on references valid IDs within same pool (K55)
1839
1840 def check_48() -> Dict:
1841     _hdr('CHECK 48: depends_on entries reference valid task IDs within same pool')
1842     from config.base_config import SCENARIO_TASK_TYPES
1843     failures = []
1844     for scenario, pool in SCENARIO_TASK_TYPES.items():
1845         valid_ids = {t['id'] for t in pool}
1846         for task in pool:
1847             for dep in task.get('depends_on', []):
1848                 if dep not in valid_ids:
1849                     failures.append(
1850                         f'{scenario}/{task["id"]}: dep "{dep}" not in pool')
1851     if failures:

```

```

1852         for f in failures:
1853             print(f' ERROR: {f}')
1854         _v('CHECK 48', 'FAIL', f'{len(failures)} invalid dependency
reference(s)')
1855         return {'verdict': 'FAIL', 'failures': failures}
1856     _v('CHECK 48', 'PASS', 'all depends_on entries reference valid IDs')
1857     return {'verdict': 'PASS'}
1858
1859
1860 # CHECK 49 - DEPENDENT_TASK_REWARD_MULTIPLIER == 1.5 (K56)
1861
1862 def check_49() -> Dict:
1863     _hdr('CHECK 49: DEPENDENT_TASK_REWARD_MULTIPLIER == 1.5')
1864     from config.base_config import DEPENDENT_TASK_REWARD_MULTIPLIER as _M
1865     print(f' DEPENDENT_TASK_REWARD_MULTIPLIER = {_M}')
1866     if _M == 1.5:
1867         _v('CHECK 49', 'PASS', 'multiplier is 1.5')
1868         return {'verdict': 'PASS'}
1869     _v('CHECK 49', 'FAIL', f'expected 1.5, got {_M}')
1870     return {'verdict': 'FAIL', 'value': _M}
1871
1872
1873 # CHECK 50 - ×1.5 multiplier wired into _complete_task (K57)
1874
1875 def check_50() -> Dict:
1876     _hdr('CHECK 50: ×1.5 reward multiplier wired into _complete_task')
1877     src = Path('environment/task_env.py').read_text(encoding='utf-8')
1878     uses = 'DEPENDENT_TASK_REWARD_MULTIPLIER' in src
1879     idx = src.find('def _complete_task')
1880     next_def = src.find('\n    def ', idx + 1) if idx >= 0 else -1
1881     func_src = (src[idx:next_def] if next_def > 0 else src[idx:idx +
3000]) \
1882         if idx >= 0 else ''
1883     in_func = 'DEPENDENT_TASK_REWARD_MULTIPLIER' in func_src

```



```

1884     print(f'  DEPENDENT_TASK_REWARD_MULTIPLIER in task_env.py : {uses}')
1885     print(f'  used inside _complete_task                :
1886           {in_func}')
1886     if uses and in_func:
1887         _v('CHECK 50', 'PASS', '×1.5 multiplier wired into
1888           _complete_task')
1888         return {'verdict': 'PASS'}
1889     if uses:
1890         _v('CHECK 50', 'AMBIG',
1891           'multiplier defined but not found inside _complete_task')
1892         return {'verdict': 'AMBIG'}
1893     _v('CHECK 50', 'FAIL',
1894       'DEPENDENT_TASK_REWARD_MULTIPLIER not found in task_env.py')
1895     return {'verdict': 'FAIL'}
1896
1897
1898 # CHECK 51 - task type IDs unique within each pool (K58)
1899
1900 def check_51() -> Dict:
1901     _hdr('CHECK 51: task type IDs are unique within each pool')
1902     from config.base_config import SCENARIO_TASK_TYPES
1903     from collections import Counter
1904     failures = []
1905     for scenario, pool in SCENARIO_TASK_TYPES.items():
1906         ids = [t['id'] for t in pool]
1907         dupes = [tid for tid, n in Counter(ids).items() if n > 1]
1908         ok = not dupes
1909         c = _GREEN if ok else _RED
1910         print(f' {c}{scenario:<22}: {len(ids)} IDs, dupes={dupes}{_END}')
1911         if not ok:
1912             failures.append(f'{scenario}: {dupes}')
1913     if failures:
1914         _v('CHECK 51', 'FAIL', f'duplicate IDs: {failures}')
1915     return {'verdict': 'FAIL', 'failures': failures}

```

```

1916     _v('CHECK 51', 'PASS', 'all task type IDs unique within their pools')
1917     return {'verdict': 'PASS'}
1918
1919
1920 # CHECK 52 - obs shape == (FLAT_OBS_DIM,) (K61)
1921
1922 def check_52(scenario: str = 'warehouse') -> Dict:
1923     _hdr(f'CHECK 52: env obs shape == (FLAT_OBS_DIM={FLAT_OBS_DIM},)
1924         ({scenario})')
1925     try:
1926         env = ImprovedTaskAllocationEnv(scenario=scenario)
1927         obs, _ = env.reset(seed=42)
1928     except Exception as e:
1929         _v('CHECK 52', 'FAIL', str(e))
1930         return {'verdict': 'FAIL', 'error': str(e)}
1931     print(f'  obs shape      : {obs.shape}')
1932     print(f'  FLAT_OBS_DIM : {FLAT_OBS_DIM}')
1933     if obs.shape == (FLAT_OBS_DIM,):
1934         _v('CHECK 52', 'PASS', f'obs shape = ({FLAT_OBS_DIM},)')
1935         return {'verdict': 'PASS'}
1936     _v('CHECK 52', 'FAIL', f'expected ({FLAT_OBS_DIM},), got {obs.shape}')
1937     return {'verdict': 'FAIL', 'shape': list(obs.shape)}
1938
1939 # CHECK 53 - MAPPO get_action with N=2 robots (K62)
1940
1941 def check_53() -> Dict:
1942     _hdr('CHECK 53: MAPPO get_action works with N=2 robots')
1943     from models.rl_agents import MAPPO
1944     from config.base_config import LOCAL_OBS_DIM, GLOBAL_OBS_DIM,
1945         ACTION_DIM
1946     agent = MAPPO()
1947     local_obs = torch.randn(2, LOCAL_OBS_DIM)
1948     global_obs = torch.randn(2, GLOBAL_OBS_DIM)

```

```

1948     try:
1949         with torch.no_grad():
1950             action, log_prob, value = agent.get_action(local_obs,
1951                 global_obs)
1952     except Exception as e:
1953         _v('CHECK 53', 'FAIL', f'{type(e).__name__}: {e}')
1954         return {'verdict': 'FAIL', 'error': str(e)}
1955     print(f' N=2 actions={action.tolist()} log_prob
1956         shape={log_prob.shape}')
1957     in_range = all(0 <= int(a) < ACTION_DIM for a in action)
1958     if in_range:
1959         _v('CHECK 53', 'PASS', f'N=2 actions in [0, {ACTION_DIM}]')
1960         return {'verdict': 'PASS'}
1961     _v('CHECK 53', 'FAIL', f'actions {action.tolist()} out of [0,
1962         {ACTION_DIM}]')
1963     return {'verdict': 'FAIL'}
1964
1965 # CHECK 54 - MAPPO get_action with N=20 robots (K63)
1966
1967 def check_54() -> Dict:
1968     _hdr('CHECK 54: MAPPO get_action works with N=20 robots')
1969     from models.rl_agents import MAPPO
1970     from config.base_config import LOCAL_OBS_DIM, GLOBAL_OBS_DIM,
1971         ACTION_DIM
1972     agent = MAPPO()
1973     local_obs = torch.randn(20, LOCAL_OBS_DIM)
1974     global_obs = torch.randn(20, GLOBAL_OBS_DIM)
1975     try:
1976         with torch.no_grad():
1977             action, log_prob, value = agent.get_action(local_obs,
1978                 global_obs)
1979     except Exception as e:
1980         _v('CHECK 54', 'FAIL', f'{type(e).__name__}: {e}')

```

```

1977         return {'verdict': 'FAIL', 'error': str(e)}
1978     print(f'  N=20 action shape={action.shape} log_prob
1979           shape={log_prob.shape}')
1980     in_range = all(0 <= int(a) < ACTION_DIM for a in action)
1981     if in_range:
1982         _v('CHECK 54', 'PASS', f'N=20 actions in [0, {ACTION_DIM}]')
1983         return {'verdict': 'PASS'}
1984     _v('CHECK 54', 'FAIL', 'actions out of range')
1985     return {'verdict': 'FAIL'}
1986
1987 # CHECK 55 - QMIX actions in valid range for varied fleet sizes (K64)
1988
1989 def check_55() -> Dict:
1990     _hdr('CHECK 55: QMIX forward action in valid range for N=2 and N=20')
1991     from models.rl_agents import QMIX
1992     from config.base_config import LOCAL_OBS_DIM
1993     qmix = QMIX()
1994     failures = []
1995     for n in [2, 5, 20]:
1996         local_obs = torch.randn(n, LOCAL_OBS_DIM)
1997         with torch.no_grad():
1998             _, task_id = qmix(local_obs)
1999             tids = task_id.tolist() if task_id.dim() > 0 else [task_id.item()]
2000             bad = [t for t in tids if not (0 <= t < QMIX.MAX_TASKS)]
2001             ok = not bad
2002             c = _GREEN if ok else _RED
2003             print(f'  {c}N={n}<3} tids={tids[:3]} in [0,{QMIX.MAX_TASKS}):
2004                   {ok}-{_END}')
2005             if not ok:
2006                 failures.append(f'N={n}: {bad}')
2007     if failures:
2008         _v('CHECK 55', 'FAIL', '; '.join(failures))
2009     return {'verdict': 'FAIL', 'failures': failures}

```

```

2009     _v('CHECK 55', 'PASS', 'QMIX task_ids in range for N=2,5,20')
2010     return {'verdict': 'PASS'}
2011
2012
2013     # CHECK 56 - SAR env smoke
2014
2015     def check_56() -> Dict:
2016         _hdr('CHECK 56: SAR env smoke (reset + step)')
2017         return _scenario_smoke('search_and_rescue', '56')
2018
2019
2020     # CHECK 57 - Maritime env smoke
2021
2022     def check_57() -> Dict:
2023         _hdr('CHECK 57: Maritime env smoke (reset + step)')
2024         return _scenario_smoke('maritime_security', '57')
2025
2026
2027     def _scenario_smoke(scenario: str, check_num: str) -> Dict:
2028         try:
2029             env = ImprovedTaskAllocationEnv(scenario=scenario)
2030             obs, info = env.reset(seed=42)
2031             obs2, r, term, trunc, _ = env.step(env.max_robots * env.max_tasks)
2032         except Exception as e:
2033             _v(f'CHECK {check_num}', 'FAIL', f'{type(e).__name__}: {e}')
2034             traceback.print_exc()
2035             return {'verdict': 'FAIL', 'error': str(e)}
2036         print(f'  scenario   : {scenario}')
2037         print(f'  obs shape  : {obs.shape}  num_robots: {env.num_robots}')
2038         print(f'  step r     : {r:+.3f}')
2039         ok = obs.shape == obs2.shape and len(obs) > 0 and np.isfinite(r)
2040         if ok:
2041             _v(f'CHECK {check_num}', 'PASS', f'{scenario} smoke OK')
2042             return {'verdict': 'PASS'}

```

```

2043     _v(f'CHECK {check_num}', 'FAIL', 'obs inconsistent or reward not
2044         finite')
2045
2046     return {'verdict': 'FAIL'}
2047
2048     # CHECK 58 - env.step returns finite rewards
2049
2050 def check_58(scenario: str = 'warehouse', n_steps: int = 100) -> Dict:
2051     _hdr(f'CHECK 58: env.step returns finite rewards ({scenario},
2052         {n_steps} steps)')
2053
2054     try:
2055         env = ImprovedTaskAllocationEnv(scenario=scenario)
2056         env.reset(seed=42)
2057
2058     except Exception as e:
2059         _v('CHECK 58', 'FAIL', str(e))
2060         return {'verdict': 'FAIL', 'error': str(e)}
2061
2062     rng = np.random.RandomState(55)
2063     n_op = env.max_robots * env.max_tasks
2064     bad_rewards = []
2065
2066     for i in range(n_steps):
2067         try:
2068             _, r, term, trunc, _ = env.step(rng.randint(0, n_op + 1))
2069
2070         except Exception as e:
2071             _v('CHECK 58', 'FAIL', f'step {i} crashed: {e}')
2072             return {'verdict': 'FAIL', 'error': str(e)}
2073
2074         if not np.isfinite(r):
2075             bad_rewards.append((i, r))
2076
2077         if term or trunc:
2078             env.reset(seed=i)
2079
2080     print(f'  steps: {n_steps}  non-finite rewards: {len(bad_rewards)}')
2081
2082     if bad_rewards:
2083         _v('CHECK 58', 'FAIL', f'{len(bad_rewards)} non-finite reward(s)')
2084         return {'verdict': 'FAIL', 'bad': bad_rewards[:5]}
2085
2086     _v('CHECK 58', 'PASS', f'all {n_steps} rewards finite')

```

```

2075     return {'verdict': 'PASS'}
2076
2077
2078 # CHECK 59 - run.py modes and aliases resolve correctly
2079
2080 def check_59() -> Dict:
2081     _hdr('CHECK 59: run.py modes and aliases present and importable')
2082     from config.base_config import MODES
2083
2084     # These must be proper MODES entries (not aliases)
2085     required_modes = ['sanity', 'quick', 'mini', '2hr', 'full_train',
2086                     'full',
2087                     'real_world']
2088     # These must remain as _MODE_ALIASES in run.py
2089     required_aliases = ['smoke_test', 'mini_train']
2090
2091     run_src = Path('run.py').read_text(encoding='utf-8')
2092
2093     print('  MODES entries (config/base_config.py):')
2094     mode_missing = []
2095     for m in required_modes:
2096         ok = m in MODES
2097         c = _GREEN if ok else _RED
2098         print(f'    {c}{m:<18}: {"present" if ok else "MISSING"}{_END}')
2099         if not ok:
2100             mode_missing.append(m)
2101
2102     print('  _MODE_ALIASES (run.py):')
2103     alias_missing = []
2104     for a in required_aliases:
2105         ok = a in run_src
2106         c = _GREEN if ok else _RED
2107         print(f'    {c}{a:<18}: {"present" if ok else "MISSING"}{_END}')
2108         if not ok:

```

```

2108         alias_missing.append(a)
2109
2110     try:
2111         import run as _run_mod
2112         print(f' run.py imports OK: True')
2113     except Exception as e:
2114         _v('CHECK 59', 'FAIL', f'run.py import failed: {e}')
2115         return {'verdict': 'FAIL', 'error': str(e)}
2116
2117     all_missing = mode_missing + alias_missing
2118     if all_missing:
2119         _v('CHECK 59', 'FAIL', f'missing: {all_missing}')
2120         return {'verdict': 'FAIL', 'missing': all_missing}
2121     _v('CHECK 59', 'PASS',
2122         f'all {len(required_modes)} modes + {len(required_aliases)}
2123         aliases present')
2124     return {'verdict': 'PASS'}
2125
2126 # CHECK 60 - SimReadyRL forward pass
2127
2128 def check_60() -> Dict:
2129     _hdr('CHECK 60: SimReadyRL forward pass (MAPPO, QMIX, DQN ×
2130         warehouse)')
2131     from models.sim_ready_wrappers import SimReadyRL
2132     from models.rl_agents import MAPPO, QMIX, DQN
2133     obs = torch.randn(1, FLAT_OBS_DIM)
2134     failures = []
2135     for algo, model in [('MAPPO', MAPPO()), ('QMIX', QMIX()), ('DQN',
2136         DQN())]:
2137         try:
2138             w = SimReadyRL(model, algo, 'warehouse')
2139             with torch.no_grad():
2140                 robot_id, task_id = w(obs)

```



```

2139         ok = torch.is_tensor(robot_id) and torch.is_tensor(task_id)
2140         c = _GREEN if ok else _RED
2141         print(f' {c}{algo:<8}: robot={int(robot_id)},
task={int(task_id)}{_END}')
2142         if not ok:
2143             failures.append(algo)
2144         except Exception as e:
2145             print(f' {_RED}{algo:<8}: FAIL --- {e}{_END}')
2146             failures.append(f'{algo}: {e}')
2147     if failures:
2148         _v('CHECK 60', 'FAIL', f'{len(failures)} model(s) failed')
2149         return {'verdict': 'FAIL', 'failures': failures}
2150     _v('CHECK 60', 'PASS', 'MAPPO/QMIX/DQN SimReadyRL forward OK')
2151     return {'verdict': 'PASS'}
2152
2153
2154 # CHECK 61 - obs values all finite after reset
2155
2156 def check_61(scenario: str = 'warehouse') -> Dict:
2157     _hdr(f'CHECK 61: obs has no NaN or Inf after env reset ({scenario})')
2158     try:
2159         env = ImprovedTaskAllocationEnv(scenario=scenario)
2160         obs, _ = env.reset(seed=42)
2161     except Exception as e:
2162         _v('CHECK 61', 'FAIL', str(e))
2163         return {'verdict': 'FAIL', 'error': str(e)}
2164     nan_count = int(np.sum(~np.isfinite(obs)))
2165     print(f'  obs shape   : {obs.shape}')
2166     print(f'  nan/inf      : {nan_count}')
2167     print(f'  range        : [{obs.min():.3f}, {obs.max():.3f}']')
2168     if nan_count == 0:
2169         _v('CHECK 61', 'PASS', 'obs is all finite after reset')
2170         return {'verdict': 'PASS'}
2171     _v('CHECK 61', 'FAIL', f'{nan_count} non-finite values in obs')

```

```

2172     return {'verdict': 'FAIL', 'nan_count': nan_count}
2173
2174
2175 # CHECK 62 - build_action_mask: AVAILABLE tasks are valid (not masked)
2176
2177 def check_62() -> Dict:
2178     _hdr('CHECK 62: build_action_mask --- AVAILABLE tasks are valid
2179         (mask=False)')
2180
2181     from utils.action_masking import build_action_mask
2182     from config.base_config import TASK_STATUS_AVAILABLE,
2183     TASK_STATUS_UNAVAILABLE
2184
2185     tasks = [
2186         {'status': TASK_STATUS_AVAILABLE, 'dependencies': []},
2187         {'status': TASK_STATUS_UNAVAILABLE, 'dependencies': [0]},
2188         {'status': TASK_STATUS_AVAILABLE, 'dependencies': []},
2189     ]
2190
2191     mask = build_action_mask(tasks, num_robots=1, max_tasks=3)
2192
2193     t0_valid = not bool(mask[0].item()) # AVAILABLE → should be
2194     False (valid)
2195
2196     t1_invalid = bool(mask[1].item()) # UNAVAILABLE → True
2197     (invalid)
2198
2199     t2_valid = not bool(mask[2].item()) # AVAILABLE → False (valid)
2200
2201     print(f' task 0 AVAILABLE valid : {t0_valid}')
2202     print(f' task 1 UNAVAILABLE invalid: {t1_invalid}')
2203     print(f' task 2 AVAILABLE valid : {t2_valid}')
2204
2205     if t0_valid and t1_invalid and t2_valid:
2206         _v('CHECK 62', 'PASS', 'AVAILABLE tasks correctly unmasked')
2207         return {'verdict': 'PASS'}
2208
2209     _v('CHECK 62', 'FAIL',
2210         'AVAILABLE task masked or UNAVAILABLE task unmasked')
2211     return {'verdict': 'FAIL'}
2212
2213
2214 # CHECK 63 - N=2 fleet completes tasks over multiple episodes

```

```

2202
2203 def check_63(scenario: str = 'warehouse', n_eps: int = 5) -> Dict:
2204     _hdr(f'CHECK 63: N=2 fleet completes  $\geq 1$  task over {n_eps} episodes
        ({scenario})')
2205     from config.base_config import TASK_STATUS_AVAILABLE
2206     total_completions = 0
2207     try:
2208         for ep in range(n_eps):
2209             env = _build_env(scenario, num_robots=2, num_tasks=2,
max_steps=500)
2210             env.reset(seed=ep * 7)
2211             n_op = env.max_robots * env.max_tasks
2212             rng = np.random.RandomState(ep)
2213             for _ in range(500):
2214                 valid = [r * env.max_tasks + t
2215                         for r in range(env.num_robots)
2216                         for t in range(env.num_tasks)
2217                         if env.tasks[t].get('status') ==
TASK_STATUS_AVAILABLE
2218                             and t not in env.task_assignments.values()]
2219                 a = int(rng.choice(valid)) if valid else n_op
2220                 _, _, term, trunc, _ = env.step(a)
2221                 if term or trunc:
2222                     break
2223                 total_completions += len(env.completed_tasks)
2224     except Exception as e:
2225         _v('CHECK 63', 'FAIL', str(e))
2226         return {'verdict': 'FAIL', 'error': str(e)}
2227     avg = total_completions / n_eps
2228     print(f' episodes: {n_eps} total completions: {total_completions}
        avg: {avg:.2f}')
2229     if total_completions > 0:
2230         _v('CHECK 63', 'PASS',
2231             f'N=2 fleet completed {total_completions} tasks over {n_eps}

```

```

    episodes')
2232     return {'verdict': 'PASS', 'total': total_completions}
2233     _v('CHECK 63', 'FAIL', 'no task completions with N=2 fleet')
2234     return {'verdict': 'FAIL', 'total': 0}
2235
2236
2237 # CHECK 64 - augment flip-y negates y-coordinates
2238
2239 def check_64() -> Dict:
2240     _hdr('CHECK 64: augment flip-y negates y-coordinates')
2241     from training.diffusion_trainer import TrajectoryDataset
2242     wps = [[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]]
2243     base = [{'obs': [0.0], 'start_pos': [1.0, 0.5],
2244             'goal_pos': [2.0, 3.0], 'waypoints': wps}]
2245     aug = TrajectoryDataset.augment(base)
2246     flip_y = aug[2] # index 2 = flip-y variant
2247     orig = np.array(wps)
2248     fwp = np.array(flip_y['waypoints'])
2249     x_ok = np.allclose(fwp[:, 0], orig[:, 0])
2250     y_ok = np.allclose(fwp[:, 1], -orig[:, 1])
2251     sy_ok = flip_y['start_pos'][1] == -base[0]['start_pos'][1]
2252     print(f' orig y: {orig[:, 1].tolist()}')
2253     print(f' flip y: {fwp[:, 1].tolist()}')
2254     print(f' y negated={y_ok} x unchanged={x_ok} start_y ok={sy_ok}')
2255     if x_ok and y_ok and sy_ok:
2256         _v('CHECK 64', 'PASS', 'flip-y correctly negates y-coordinates')
2257         return {'verdict': 'PASS'}
2258     _v('CHECK 64', 'FAIL', f'flip-y incorrect: y_ok={y_ok}, x_ok={x_ok}')
2259     return {'verdict': 'FAIL'}
2260
2261
2262 # CHECK 65 - MODES dict has all expected mode keys
2263
2264 def check_65() -> Dict:

```

```

2265     _hdr('CHECK 65: config/base_config.py MODES has all required mode
keys')
2266     from config.base_config import MODES
2267     required = ['sanity', 'quick', 'mini', '2hr', 'full_train', 'full',
'real_world']
2268     required_fields = ['rl_episodes', 'diff_epochs', 'max_steps',
'num_workers',
2269                       'rl_batch_size', 'diff_batch_size', 'log_every']
2270     missing_modes = []
2271     missing_fields = []
2272     for mode in required:
2273         if mode not in MODES:
2274             missing_modes.append(mode)
2275             print(f'  {_RED}{mode:<18}: MISSING{_END}')
2276             continue
2277         m = MODES[mode]
2278         mf = [f for f in required_fields if f not in m]
2279         if mf:
2280             missing_fields.append(f'{mode}: {mf}')
2281         c = _GREEN if not mf else _YEL
2282         print(f'  {c}{mode:<18}: present '
2283               f' (ep={m.get("rl_episodes")}, steps={m.get("max_steps")}, '
2284               f'workers={m.get("num_workers")}){_END}')
2285
2286     if missing_modes:
2287         _v('CHECK 65', 'FAIL', f'missing modes: {missing_modes}')
2288         return {'verdict': 'FAIL', 'missing_modes': missing_modes}
2289     if missing_fields:
2290         _v('CHECK 65', 'AMBIG', f'modes with missing fields:
{missing_fields}')
2291         return {'verdict': 'AMBIG', 'missing_fields': missing_fields}
2292     _v('CHECK 65', 'PASS',
2293       f'all {len(required)} required modes present with correct fields')
2294     return {'verdict': 'PASS'}

```

```

2295
2296
2297 # CHECK 66 - PerformancePacedGenerator adjusts difficulty correctly
2298
2299 def check_66() -> Dict:
2300     _hdr('CHECK 66: PerformancePacedGenerator difficulty adjusts within
2301           thresholds')
2302     try:
2303         from environment.task_env import PerformancePacedGenerator
2304     except ImportError as e:
2305         _v('CHECK 66', 'FAIL', f'PerformancePacedGenerator not
2306           importable: {e}')
2307         return {'verdict': 'FAIL', 'error': str(e)}
2308
2309     ppg = PerformancePacedGenerator('warehouse')
2310     initial = ppg._difficulty
2311
2312     # Feed HIGH completion rates (>80%) --- difficulty should increase
2313     for _ in range(25):
2314         ppg.record(0.90)
2315         after_high = ppg._difficulty
2316         increased = after_high > initial
2317
2318     # Reset and feed LOW completion rates (<40%) --- difficulty should
2319     decrease
2320     ppg2 = PerformancePacedGenerator('warehouse')
2321     ppg2._difficulty = 0.5
2322     before_low = ppg2._difficulty
2323     for _ in range(25):
2324         ppg2.record(0.10)
2325         after_low = ppg2._difficulty
2326         decreased = after_low < before_low
2327
2328     params = ppg.get_params()

```

```

2326     has_num_robots = 'num_robots_range' in params
2327     has_num_tasks  = 'num_tasks_range' in params
2328
2329     print(f'  initial difficulty : {initial:.3f}')
2330     print(f'  after 25× 90%      : {after_high:.3f}
      (increased={increased})')
2331     print(f'  before low test   : {before_low:.3f}')
2332     print(f'  after 25× 10%     : {after_low:.3f}
      (decreased={decreased})')
2333     print(f'  get_params keys   : {sorted(params.keys())}')
2334     print(f'  has num_robots_range: {has_num_robots}')
2335     print(f'  has num_tasks_range : {has_num_tasks}')
2336
2337     if increased and decreased and has_num_robots and has_num_tasks:
2338         _v('CHECK 66', 'PASS',
2339           'PPG increases difficulty on high success, decreases on low
      success')
2340         return {'verdict': 'PASS',
2341               'after_high': after_high, 'after_low': after_low}
2342     issues = []
2343     if not increased:
2344         issues.append(f'difficulty did not increase after high completion
      ,
2345                       f'({initial:.3f} -> {after_high:.3f})')
2346     if not decreased:
2347         issues.append(f'difficulty did not decrease after low completion '
2348                       f'({before_low:.3f} -> {after_low:.3f})')
2349     if not has_num_robots:
2350         issues.append('get_params missing num_robots_range')
2351     if not has_num_tasks:
2352         issues.append('get_params missing num_tasks_range')
2353     _v('CHECK 66', 'FAIL', ', '.join(issues))
2354     return {'verdict': 'FAIL', 'issues': issues}
2355

```

```

2356
2357 # CHECK 67 - SCENARIO_CONFIGS grid sizes match realistic values
2358
2359 def check_67() -> Dict:
2360     _hdr('CHECK 67: SCENARIO_CONFIGS grid_half and max_steps match
2361         realistic values')
2362
2363     from config.base_config import SCENARIO_CONFIGS, SCENARIO_NORM
2364
2365     expected = {
2366         'warehouse':          {'grid_half': 25.0,  'max_steps': 1000,
2367                                'min_steps': 100},
2368         'search_and_rescue': {'grid_half': 50.0,  'max_steps': 1200,
2369                                'min_steps': 300},
2370         'maritime_security': {'grid_half': 250.0, 'max_steps': 1500,
2371                                'min_steps': 600},
2372     }
2373
2374     failures = []
2375
2376     for sc, exp in expected.items():
2377         cfg = SCENARIO_CONFIGS[sc]
2378         norm = SCENARIO_NORM[sc]
2379         gh_ok = cfg.grid_half == exp['grid_half']
2380         ms_ok = exp['min_steps'] <= cfg.max_steps <= exp['max_steps']
2381         ng_ok = norm['grid_half'] == cfg.grid_half
2382         row_ok = gh_ok and ms_ok and ng_ok
2383         c = _GREEN if row_ok else _RED
2384         print(
2385             f' {c}{sc:<22}: grid_half={cfg.grid_half} ',
2386             f'(want {exp["grid_half"]}, ok={gh_ok}) ',
2387             f'max_steps={cfg.max_steps} ',
2388             f'(want {exp["min_steps"]}-{exp["max_steps"]}, ok={ms_ok}) ',
2389             f'norm_match={ng_ok}-{_END}'
2390         )
2391         if not row_ok:

```



```

2386         failures.append(f'{sc}: grid_half={cfg.grid_half}, '
2387                        f'max_steps={cfg.max_steps},
norm_gh={norm["grid_half"]}')
2388     if failures:
2389         _v('CHECK 67', 'FAIL', f'{len(failures)} scenario(s) with wrong
config')
2390         return {'verdict': 'FAIL', 'failures': failures}
2391     _v('CHECK 67', 'PASS', 'all scenario grid_half and max_steps values
correct')
2392     return {'verdict': 'PASS'}
2393
2394
2395 # CHECK 68 - get_scenario_action_mask accepts obs_radius_m / comm_range_m
2396
2397 def check_68() -> Dict:
2398     _hdr('CHECK 68: get_scenario_action_mask accepts obs_radius_m /
comm_range_m params')
2399     import inspect
2400     from utils.action_masking import get_scenario_action_mask
2401     from config.base_config import SCENARIO_CONFIGS
2402
2403     sig = inspect.signature(get_scenario_action_mask)
2404     params = set(sig.parameters.keys())
2405     has_obs = 'obs_radius_m' in params
2406     has_comm = 'comm_range_m' in params
2407     print(f' signature params : {sorted(params)}')
2408     print(f' has obs_radius_m : {has_obs}')
2409     print(f' has comm_range_m : {has_comm}')
2410
2411     if not (has_obs and has_comm):
2412         _v('CHECK 68', 'FAIL',
2413            'obs_radius_m and/or comm_range_m params missing from
signature')
2414         return {'verdict': 'FAIL',

```

```

2415         'missing': [p for p in ['obs_radius_m', 'comm_range_m']
2416                        if p not in params]}
2417
2418     # Functional check: call all 3 scenarios with real config values
2419     failures = []
2420     for sc in ('warehouse', 'search_and_rescue', 'maritime_security'):
2421         cfg = SCENARIO_CONFIGS[sc]
2422         try:
2423             n_tasks    = 5
2424             n_robots   = 3
2425             t_states   = np.ones(n_tasks, dtype=int)
2426             r_assign   = np.full(n_robots, -1, dtype=int)
2427             r_pos      = np.random.randn(n_robots, 2) * 10.0
2428             t_pos      = np.random.randn(n_tasks, 2) * 10.0
2429             r_carry    = np.zeros(n_robots, dtype=bool)
2430             t_pri      = np.ones(n_tasks, dtype=int) * 3
2431             mask = get_scenario_action_mask(
2432                 sc, n_tasks, t_states, r_assign, r_pos, t_pos, r_carry,
2433                 t_pri,
2434                 obs_radius_m=cfg.observation_radius,
2435                 comm_range_m=cfg.communication_range,
2436                 )
2437             expected_shape = (n_robots, n_tasks + 1)
2438             ok = mask.shape == expected_shape
2439             c = _GREEN if ok else _RED
2440             print(f' {c}{sc:<22}: shape={mask.shape} '
2441                   f'(want {expected_shape}, ok={ok}){_END}')
2442             if not ok:
2443                 failures.append(f'{sc}: got shape {mask.shape}')
2444         except Exception as e:
2445             print(f' {_RED}{sc}: ERROR --- {e}{_END}')
2446             failures.append(f'{sc}: {type(e).__name__}: {e}')
2447
2448     if failures:

```

```

2448         _v('CHECK 68', 'FAIL', f'{len(failures)} scenario(s) failed:
{failures}'))
2449         return {'verdict': 'FAIL', 'failures': failures}
2450     _v('CHECK 68', 'PASS',
2451         'get_scenario_action_mask accepts obs_radius_m/comm_range_m and '
2452         'returns correct shapes for all 3 scenarios')
2453     return {'verdict': 'PASS'}
2454
2455
2456     # Per-check timeout helper
2457
2458     def _timed_run(fn, timeout_s: float, *args, **kwargs) -> Dict:
2459         """Run fn(*args, **kwargs) in a thread with timeout_s wall-clock
limit."""
2460         with concurrent.futures.ThreadPoolExecutor(max_workers=1) as ex:
2461             fut = ex.submit(fn, *args, **kwargs)
2462             try:
2463                 return fut.result(timeout=timeout_s)
2464             except concurrent.futures.TimeoutError:
2465                 return {'verdict': 'FAIL',
2466                     'error': f'timed out after {timeout_s:.0f}s'}
2467             except Exception as e:
2468                 return {'verdict': 'CRASH', 'error': str(e)}
2469
2470
2471     _CHECKS = [
2472         # Existing checks 1--11
2473         ('1', check_1, False), ('2', check_2, False), ('3', check_3,
False),
2474         ('4', check_4, False), ('5', check_5, False), ('6', check_6,
False),
2475         ('7', check_7, False), ('8', check_8, True),
2476         ('9', check_9, False), ('10', check_10, False), ('11', check_11,
False),

```

```

2477      # New checks 12--18: task status state machine (K8--K14)
2478      ('12', check_12, False), ('13', check_13, False), ('14', check_14,
2479      False),
2480      ('15', check_15, False), ('16', check_16, False), ('17', check_17,
2481      False),
2482      ('18', check_18, False),
2483      # New checks 19--21: battery / deactivation (K18--K20)
2484      ('19', check_19, False), ('20', check_20, False), ('21', check_21,
2485      False),
2486      # New checks 22--30: hex grid (K21--K29)
2487      ('22', check_22, False), ('23', check_23, False), ('24', check_24,
2488      False),
2489      ('25', check_25, False), ('26', check_26, False), ('27', check_27,
2490      False),
2491      ('28', check_28, False), ('29', check_29, False), ('30', check_30,
2492      False),
2493      # New checks 31--38: action masking + RL pipeline (K34--K41)
2494      ('31', check_31, False), ('32', check_32, False), ('33', check_33,
2495      False),
2496      ('34', check_34, False), ('35', check_35, False), ('36', check_36,
2497      False),
2498      ('37', check_37, False), ('38', check_38, False),
2499      # New checks 39--44: diffusion pipeline (K46--K51)
2500      ('39', check_39, False), ('40', check_40, False), ('41', check_41,
2501      False),
2502      ('42', check_42, False), ('43', check_43, False), ('44', check_44,
2503      False),
2504      # New checks 45--51: task type tables (K52--K58)
2505      ('45', check_45, False), ('46', check_46, False), ('47', check_47,
2506      False),
2507      ('48', check_48, False), ('49', check_49, False), ('50', check_50,
2508      False),
2509      ('51', check_51, False),
2510      # New checks 52--55: scalable policy (K61--K64)

```

```

2499     ('52', check_52, False), ('53', check_53, False), ('54', check_54,
2500     False),
2501     ('55', check_55, False),
2502     # New checks 56--64: scenario smokes + misc correctness
2503     ('56', check_56, False), ('57', check_57, False), ('58', check_58,
2504     False),
2505     ('59', check_59, False), ('60', check_60, False), ('61', check_61,
2506     False),
2507     ('62', check_62, False), ('63', check_63, False), ('64', check_64,
2508     False),
2509     # New checks 65--68: mode dict, PPG, grid sizes, masking params
2510     ('65', check_65, False), ('66', check_66, False),
2511     ('67', check_67, False), ('68', check_68, False),
2512 ]
2513
2514 def main() -> int:
2515     n_checks = len(_CHECKS)
2516     _all_check_nums = ['all'] + [str(i) for i in range(1, n_checks + 1)]
2517     p = argparse.ArgumentParser(
2518         description=f'AQUA Guardian unified sanity script ({n_checks}
2519         checks).',
2520         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
2521     p.add_argument('--check', default='all', choices=_all_check_nums,
2522         help='run a single check by number, or "all"')
2523     p.add_argument('--quick', action='store_true',
2524         help='skip slow checks (currently just check 8)')
2525     p.add_argument('--scenario', default='warehouse',
2526         choices=['warehouse', 'search_and_rescue',
2527         'maritime_security'])
2528     p.add_argument('--episodes', type=int, default=1000,
2529         help='check 8: number of episodes')
2530     p.add_argument('--max_steps', type=int, default=200,
2531         help='checks 6, 8: episode horizon (default 200)')

```

```

2528     p.add_argument('--timeout', type=int, default=0,
2529                    help='per-check wall-clock timeout in seconds
(0=unlimited)')
2530     args = p.parse_args()
2531
2532     total_start = time.time()
2533     print(f'{{_BOLD}}{{_BLUE}}AQUA Guardian sanity script --- {{n_checks}}
checks{{_END}}')
2534     print(f'  scenario   : {{args.scenario}}')
2535     print(f'  timeout    : {{args.timeout}}s per check (0=unlimited)')
2536     print(f'  cwd        : {{os.getcwd()}}')
2537     print(f'  python     : {{sys.version.split()[0]}}')
2538     print(f'  torch      : {{torch.__version__}}
(cuda={{torch.cuda.is_available()}})')
2539
2540     summary: Dict[str, Dict] = {}
2541     timings: Dict[str, float] = {}
2542
2543     def _run(name: str, fn, *a, **kw):
2544         t0 = time.time()
2545         try:
2546             if args.timeout > 0:
2547                 r = _timed_run(fn, float(args.timeout), *a, **kw)
2548             else:
2549                 r = fn(*a, **kw)
2550             summary[name] = r
2551         except Exception as e:
2552             print(f'\n{{_RED}}[CHECK {name}] crashed: {{e}}{{_END}}')
2553             traceback.print_exc()
2554             summary[name] = {'verdict': 'CRASH', 'error': str(e)}
2555             timings[name] = time.time() - t0
2556
2557     targets = ([ (name, fn) for name, fn, slow in _CHECKS
2558                  if not (args.quick and slow)]

```

```

2559         if args.check == 'all'
2560         else [(name, fn) for name, _ in _CHECKS
2561                if name == args.check])
2562
2563     for name, fn in targets:
2564         sig = inspect.signature(fn)
2565         has_scenario = 'scenario' in sig.parameters
2566         if name == '6':
2567             sc_cfg = SCENARIO_CONFIGS[args.scenario]
2568             _run(name, fn, scenario=args.scenario,
2569                  num_robots=sc_cfg.num_robots, max_steps=args.max_steps)
2570         elif name == '8':
2571             _run(name, fn, scenario=args.scenario,
2572                  num_episodes=args.episodes, max_steps=args.max_steps)
2573         elif has_scenario:
2574             _run(name, fn, scenario=args.scenario)
2575         else:
2576             _run(name, fn)
2577
2578     _hdr('SUMMARY')
2579     print(f'  {"Check":<10} {"Result":<8} {"Time":>7}  Details')
2580     print('  ' + '-' * 72)
2581     for name, _, _ in _CHECKS:
2582         if name not in summary:
2583             continue
2584         v    = summary[name].get('verdict', '?')
2585         err  = summary[name].get('error', '')
2586         dt   = timings.get(name, 0.0)
2587         c    = {'PASS': _GREEN, 'FAIL': _RED, 'AMBIG': _YEL,
2588                'CRASH': _RED}.get(v, _BLUE)
2589         detail = (f' --- {str(err)[:60]} ' if err and v != 'PASS' else '')
2590         print(f'  {_BOLD}CHECK {name:<4}{_END} '
2591               f'{c}{_BOLD}{v:<8}{_END} '
2592               f'{dt:5.1f}s {detail}')

```

```

2593
2594     fails = [n for n, r in summary.items()
2595               if r.get('verdict') in ('FAIL', 'CRASH')]
2596     passes = [n for n, r in summary.items() if r.get('verdict') == 'PASS']
2597     ambigs = [n for n, r in summary.items() if r.get('verdict') ==
2598               'AMBIG']
2599
2600     total_elapsed = time.time() - total_start
2601
2602     print()
2603     print(f' Total: {len(passes)} PASS {len(fails)} FAIL '
2604           f'{len(ambigs)} AMBIG --- {total_elapsed:.1f}s elapsed')
2605     print()
2606
2607     if not fails and len(passes) == len(summary):
2608         print(f'_{GREEN}_{BOLD}')
2609         'All checks passed --- env, fixes, and learning signal are
2610         healthy.'
2611         f'_{END}')
2612         return 0
2613
2614     if fails:
2615         print(f'_{RED}_{BOLD}')
2616         f'Failed: {"", ".join(fails)}. Investigate before scaling.'
2617         f'_{END}')
2618         return 1
2619
2620     print(f'_{YEL}_{BOLD}')
2621     f'No outright failures. Ambiguous: {"", ".join(ambigs)}.'
2622     f'_{END}')
2623
2624     return 0
2625
2626 if __name__ == '__main__':
2627     sys.exit(main())

```

Listing B.33: Sanity script that exercises every algorithm class and every environment type with synthetic input data, used as a pre-flight gate before full training.

```
1  """
2  fast_reserve.py
3  =====
4  Standalone GPU memory reservation script for shared-GPU server setups.
5
6  Run this in a separate terminal before starting a training job to claim
7  a fixed VRAM budget so other users cannot grab it mid-training.
8
9  Uses the same incremental burst-allocate-then-hold strategy as
10 utils/gpu_utils.py: allocate in small chunks, keep references alive,
11 never call empty_cache. 50 MB headroom left for CUDA context overhead.
12
13 Usage:
14     python fast_reserve.py
15 """
16 import torch
17 import time
18 import os
19
20 target_gb = 15
21 target_bytes = target_gb * 1024**3
22 holders = []
23 total_held_bytes = 0
24
25 print("Starting Incremental Reserver (Target: 15GB)")
26
27 while True:
28     try:
```

```

29         free, total = torch.cuda.mem_get_info()
30
31         if total_held_bytes < target_bytes and free > (10 * 1024**2):
32             to_take_now = free - (50 * 1024**2)  # 50 MB headroom for
CUDA context
33
34             # Clamp so we never overshoot the target
35             if (total_held_bytes + to_take_now) > target_bytes:
36                 to_take_now = target_bytes - total_held_bytes
37
38             if to_take_now > 0:
39                 num_elements = int(to_take_now // 4)
40                 # torch.zeros forces physical page mapping; without it the
41                 # allocation may be virtual and the driver won't report
it as used.
42                 new_block = torch.zeros((num_elements,), device='cuda')
43
44                 # Keep a reference so the block stays in PyTorch's cache
45                 holders.append(new_block)
46                 total_held_bytes += to_take_now
47
48                 current_gb = total_held_bytes / 1024**3
49                 print(f"[{time.strftime('%H:%M:%S')}] Added block. Total
Holding: {current_gb:.2f} GB")
50
51     except Exception as e:
52         print(f"Error during allocation: {e}")
53
54     time.sleep(1)

```

Listing B.34: GPU memory reservation utility used for fast-mode pre-allocation of CUDA memory at process start.

1 """

```

2 package_for_jupyter.py
3 =====
4 Bundles the training codebase into a zip for upload to Jupyter or a remote
5 training server.
6
7 Run from the project root:
8     python package_for_jupyter.py
9
10 Output: aqua_guardian_upload.zip
11
12 Only files listed in INCLUDE are packaged. run.py is the single entry
13     point;
14 separate per-mode wrapper scripts are not included.
15 """
16 import os
17 import zipfile
18 from pathlib import Path
19
20 INCLUDE = [
21     'config/base_config.py',
22     'environment/__init__.py',
23     'environment/reward_shaping.py',
24     'environment/reward_wrapper.py',
25     'environment/task_env.py',
26     'environment/env_config.py',
27     'environment/observation_extractors.py',
28     'environment/analysis.py',
29     'models/__init__.py',
30     'models/attention.py',
31     'models/rl_agents.py',
32     'models/diffusion_models.py',
33     'models/sim_ready_wrappers.py',
34     'training/__init__.py',

```

```

35     'training/rl_trainer.py',
36     'training/diffusion_trainer.py',
37     'training/statistical_eval.py',
38     'training/visualizer.py',
39     'parallel/__init__.py',
40     'parallel/job_runner.py',
41     'utils/__init__.py',
42     'utils/bspline.py',
43     'utils/noise_schedule.py',
44     'utils/action_masking.py',
45     'utils/reproducibility.py',
46     'run.py',
47     'download_deps.py',
48     'package_for_jupyter.py',
49     'requirements.txt',
50     'README.md',
51     'instructions.md',
52 ]
53
54 # Auto-discover __init__.py files
55 for pkg in ['config', 'environment', 'models', 'training', 'parallel',
56            'utils']:
57     f = f'{pkg}/__init__.py'
58     if f not in INCLUDE:
59         INCLUDE.append(f)
60
61 def package():
62     root = Path(__file__).parent
63     zname = 'aqua_guardian_upload.zip'
64
65     with zipfile.ZipFile(zname, 'w', zipfile.ZIP_DEFLATED) as zf:
66         for rel in INCLUDE:
67             full = root / rel

```

```

68         if full.exists():
69             zf.write(full, f'aqua_guardian/{rel}')
70             print(f'  + {rel}')
71         else:
72             print(f'  - MISSING: {rel}')
73
74     size_kb = os.path.getsize(zname) / 1024
75     print(f'\nCreated {zname}  ({size_kb:.1f} KB)')
76     print('\nOn the server:')
77     print('  unzip aqua_guardian_upload.zip')
78     print('  cd aqua_guardian')
79     print('  python download_deps.py          # setup check')
80     print('  python run.py --mode quick          # 5-10 min smoke test')
81
82
83 if __name__ == '__main__':
84     package()

```

Listing B.35: Project packaging script that bundles the framework into a Jupyter-friendly tarball for remote experimentation.

Appendix C

Simulation Code

This appendix lists the complete simulation codebase used to deploy the trained policies in a physical Gazebo Harmonic simulation environment. The stack provides a five-page PyQt6 wizard (`launch_sim_gui.py`) that collects configuration, generates a Gazebo SDF world file, and launches a ROS 2 Jazzy session. A separate analytics dashboard subscribes to `/sim/*` topics and renders a live 2-D top-down map, task allocation table, per-robot status cards, and scrollable metrics strip.

Source listings appear in package order: the top-level entry point first, then the `core` package (world generation, model loading, session control), the `ros2` package (nodes and launch files), the `analytics` package (dashboard and ROS listener thread), the `analytics.widgets` sub-package (four live-display widgets), the `gui` package (application bootstrap and five wizard pages), and finally the export utility under `scripts`.

Top-level entry point

```
1  """Entry point: adds project root to sys.path and launches the GUI
   wizard."""
2  from __future__ import annotations
3
4  import sys
5  from pathlib import Path
6
7  # Ensure project root is importable before any Simulation.* imports
8  _PROJ_ROOT = Path(__file__).resolve().parent.parent
9  if str(_PROJ_ROOT) not in sys.path:
10     sys.path.insert(0, str(_PROJ_ROOT))
11
12  from Simulation.gui.app import SimApp
13
14
```

```

15 def main() -> None:
16     """Create the QApplication, show the wizard, and block until exit."""
17     app = SimApp(sys.argv)
18     sys.exit(app.run())
19
20
21 if __name__ == "__main__":
22     main()

```

Listing C.1: Simulation launcher entry point. Adds the project root to `sys.path` so `Simulation.*` imports resolve, then creates `SimApp` and blocks until the wizard window is closed.

Package: core

Listing C.2: Package init for the core package.

```

1  """Mirrors training env generator to build Gazebo worlds from sim
   config."""
2  from __future__ import annotations
3
4  import logging
5  import os
6  import sys
7  from pathlib import Path
8  from typing import Any, Dict
9
10 import numpy as np
11
12 logger = logging.getLogger(__name__)
13
14 # Add project root to path so we can import training code directly

```



```

15 | _SIM_DIR   = Path(__file__).resolve().parent.parent
16 | _PROJ_ROOT = _SIM_DIR.parent
17 | if str(_PROJ_ROOT) not in sys.path:
18 |     sys.path.insert(0, str(_PROJ_ROOT))
19 |
20 | from environment.env_config import DatasetConfig, dataset_config
21 | from environment.task_env import EnvironmentGenerator,
    |     ImprovedTaskAllocationEnv
22 | from config.base_config import SCENARIO_CONFIGS, SCENARIO_TASK_TYPES
23 | from Simulation.core.world_writer import WorldWriter
24 |
25 |
26 | def build_world(sim_config: Dict[str, Any]) -> Path:
27 |     """Build a Gazebo .world SDF file from sim_config dict.
28 |
29 |     Args:
30 |         sim_config: Dict produced by the GUI wizard (all pages merged).
31 |
32 |     Returns:
33 |         Path to the generated .world file.
34 |     """
35 |     scenario = sim_config.get("scenario", "warehouse")
36 |     seed      = sim_config.get("seed", -1)
37 |
38 |     if seed >= 0:
39 |         np.random.seed(seed)
40 |
41 |     env_gen = EnvironmentGenerator(dataset_config)
42 |     env_data = env_gen.generate_environment(scenario)
43 |
44 |     # Override counts from GUI params if user specified them
45 |     num_robots = sim_config.get("num_robots",
    |                               SCENARIO_CONFIGS[scenario].num_robots)
46 |     num_tasks  = sim_config.get("num_tasks",

```

```

SCENARIO_CONFIGS[scenario].max_tasks // 2)
47
48     sc_cfg      = SCENARIO_CONFIGS[scenario]
49     grid_half = sc_cfg.grid_half
50
51     from environment.env_config import dataset_config as dc
52     obstacles = env_data.get("obstacles", np.empty((0, 3)))
53     grid_size = env_data["grid_size"]
54
55     robot_positions: list = []
56     for _ in range(num_robots):
57         for _attempt in range(30):
58             pos = np.array([
59                 np.random.uniform(-grid_size[0] / 2 + 2, grid_size[0] / 2
60                 - 2),
61                 np.random.uniform(-grid_size[1] / 2 + 2, grid_size[1] / 2
62                 - 2),
63             ])
64             clear = True
65             if isinstance(obstacles, np.ndarray) and obstacles.ndim == 2:
66                 for obs in obstacles:
67                     if np.linalg.norm(pos - obs[:2]) < float(obs[2]) +
68                     2.0:
69                         clear = False
70                         break
71             if clear:
72                 break
73             robot_positions.append(pos.tolist())
74
75     task_positions = [
76         [
77             float(np.random.uniform(-grid_size[0] / 2 + 2, grid_size[0] /
78             2 - 2)),
79             float(np.random.uniform(-grid_size[1] / 2 + 2, grid_size[1] /

```

```

2 - 2)),
76     ]
77     for _ in range(num_tasks)
78 ]
79
80 static_obstacles: list = []
81 if isinstance(obstacles, np.ndarray) and obstacles.ndim == 2:
82     num_obs = sim_config.get("num_static_obstacles", len(obstacles))
83     for obs in obstacles[:num_obs]:
84         static_obstacles.append({
85             "x": float(obs[0]),
86             "y": float(obs[1]),
87             "r": float(obs[2]),
88         })
89
90 dynamic_obstacles: list = []
91 if sim_config.get("use_dynamic_obstacles", True):
92     n_dyn = sim_config.get("num_dynamic_obstacles", 3)
93     spd   = sim_config.get("dynamic_obstacle_speed", 0.3)
94     for _ in range(n_dyn):
95         angle = float(np.random.uniform(0, 2 * np.pi))
96         dynamic_obstacles.append({
97             "x": float(np.random.uniform(-grid_size[0] / 2,
98 grid_size[0] / 2)),
99             "y": float(np.random.uniform(-grid_size[1] / 2,
100 grid_size[1] / 2)),
101             "vx": spd * float(np.cos(angle)),
102             "vy": spd * float(np.sin(angle)),
103             "r": float(np.random.uniform(0.5, 2.0)),
104         })
105
106 world_data: Dict[str, Any] = {
107     "scenario": scenario,
108     "grid_half": grid_half,

```

```

107     "grid_size":          list(grid_size),
108     "robots":             robot_positions,
109     "tasks":              task_positions,
110     "static_obstacles":   static_obstacles,
111     "dynamic_obstacles":  dynamic_obstacles,
112     "physics_engine":     sim_config.get("physics_engine", "ode"),
113     "real_time_factor":   sim_config.get("real_time_factor", 1.0),
114     "view_mode":          sim_config.get("view_mode", "3d"),
115 }
116
117 out_dir = _SIM_DIR / "gazebo" / "worlds" / "generated"
118 out_dir.mkdir(parents=True, exist_ok=True)
119 writer      = WorldWriter(world_data)
120 world_path = out_dir / f"{scenario}_{seed if seed >= 0 else
    'rand'}.world"
121 writer.write(world_path)
122 logger.info(f"Generated world: {world_path}")
123 return world_path

```

Listing C.3: World builder that mirrors the training environment generator: calls `EnvironmentGenerator` from the training codebase so Gazebo layouts sample from the same distribution the policies were trained on, then writes the SDF via `WorldWriter`.

```

1  """Load TorchScript .pt model files produced by sim_ready_wrappers.py."""
2  from __future__ import annotations
3
4  import logging
5  from pathlib import Path
6  from typing import Optional
7
8  logger = logging.getLogger(__name__)
9
10 try:

```

```

11     import torch
12     HAS_TORCH = True
13 except ImportError:
14     HAS_TORCH = False
15     logger.warning("torch not available --- model_loader will not
        function")
16
17 FLAT_OBS_DIM = 910    # from base_config
18
19
20 def load_model(path: str | Path) -> "Optional[torch.jit.ScriptModule]":
21     """Load a TorchScript model; return None on failure."""
22     if not HAS_TORCH:
23         logger.error("torch not installed --- cannot load model")
24         return None
25     try:
26         model = torch.jit.load(str(path), map_location="cpu")
27         model.eval()
28         logger.info(f"Loaded model: {path}")
29         return model
30     except Exception as exc:
31         logger.error(f"Failed to load {path}: {exc}")
32         return None
33
34
35 def validate_model(path: str | Path) -> tuple[bool, str]:
36     """Load model and run a dummy forward pass. Returns (ok, message)."""
37     if not HAS_TORCH:
38         return False, "torch not installed"
39     model = load_model(path)
40     if model is None:
41         return False, "Failed to load file"
42     try:
43         dummy_obs = torch.zeros(1, FLAT_OBS_DIM)

```

```

44     dummy_mask = torch.ones(1, 20 * 50 + 1, dtype=torch.bool)
45     with torch.no_grad():
46         robot_id, task_id = model(dummy_obs, dummy_mask)
47     return True, f"OK --- robot_id={int(robot_id)},
    task_id={int(task_id)}"
48 except Exception as exc:
49     return False, f"Forward pass failed: {exc}"

```

Listing C.4: TorchScript model loader. Wraps `torch.jit.load` behind a `HAS_TORCH` guard so the rest of the simulation can start without PyTorch installed (no-policy debug mode).

```

1  """ROS 2 node that runs trained policies and publishes robot commands."""
2  from __future__ import annotations
3
4  import logging
5  import threading
6  from typing import Any, Dict, List, Optional
7
8  import numpy as np
9
10 logger = logging.getLogger(__name__)
11
12 try:
13     import rclpy
14     from rclpy.node import Node
15     from geometry_msgs.msg import Twist
16     from nav_msgs.msg import Path as NavPath
17     from geometry_msgs.msg import PoseStamped
18     HAS_ROS = True
19 except ImportError:
20     HAS_ROS = False
21     logger.warning("rclpy not available --- robot_controller will not
    function")

```

```

22
23 try:
24     import torch
25     HAS_TORCH = True
26 except ImportError:
27     HAS_TORCH = False
28     logger.warning("torch not available --- policy inference disabled")
29
30
31 class RobotController:
32     """Manages per-robot policy inference and ROS 2 command publishing."""
33
34     def __init__(
35         self,
36         scenario: str,
37         num_robots: int,
38         model_paths: List[str],
39         grid_half: float,
40     ) -> None:
41         """Load models and prepare publisher registry."""
42         self.scenario = scenario
43         self.num_robots = num_robots
44         self.grid_half = grid_half
45         self._models: List[Any] = []
46         self._node: Optional[Any] = None
47         self._publishers: Dict[int, Any] = {}
48         self._path_pubs: Dict[int, Any] = {}
49
50         if HAS_TORCH:
51             from Simulation.core.model_loader import load_model
52             for path in model_paths:
53                 m = load_model(path)
54                 if m is not None:
55                     self._models.append(m)

```

```

56
57     def start(self, node: Any) -> None:
58         """Attach to a running rclpy node and create publishers."""
59         if not HAS_ROS:
60             logger.warning("ROS not available --- skipping publisher
creation")
61             return
62         self._node = node
63         for i in range(self.num_robots):
64             self._publishers[i] = node.create_publisher(
65                 Twist, f"/robot_{i}/cmd_vel", 10)
66             self._path_pubs[i] = node.create_publisher(
67                 NavPath, f"/sim/trajectory/{i}", 10)
68
69     def step(
70         self,
71         robot_states: List[Dict],
72         obs: np.ndarray,
73         action_mask: np.ndarray,
74     ) -> None:
75         """Run policy forward pass and publish Twist commands."""
76         if not self._models or not HAS_TORCH or not HAS_ROS:
77             return
78         import torch
79         obs_t = torch.from_numpy(obs).float().unsqueeze(0)
80         mask_t = torch.from_numpy(action_mask).bool().unsqueeze(0)
81         with torch.no_grad():
82             robot_id_t, task_id_t = self._models[0](obs_t, mask_t)
83             robot_id = int(robot_id_t.item())
84             task_id = int(task_id_t.item())
85             if robot_id < self.num_robots and self._node:
86                 cmd = Twist()
87                 cmd.linear.x = float(robot_states[robot_id].get("speed",
0.5))

```



```

88         cmd.angular.z = 0.0
89         self._publishers[robot_id].publish(cmd)

```

Listing C.5: Per-robot policy runner. Assembles the observation tensor, calls the loaded TorchScript model, and publishes the resulting action as a `geometry_msgs/Twist` command on each robot's `cmd_vel` topic.

```

1  """Orchestrates a full simulation run: env generation → launch →
    analytics."""
2  from __future__ import annotations
3
4  import logging
5  import subprocess
6  import threading
7  from pathlib import Path
8  from typing import Any, Callable, Dict, Optional
9
10 logger = logging.getLogger(__name__)
11
12 _SIM_DIR = Path(__file__).resolve().parent.parent
13
14
15 class SimSession:
16     """Manages lifecycle of a single simulation run."""
17
18     def __init__(self, config: Dict[str, Any]) -> None:
19         """Store config and initialise process handle."""
20         self.config = config
21         self._process: Optional[subprocess.Popen] = None
22         self._log_cb: Optional[Callable[[str], None]] = None
23
24     def set_log_callback(self, cb: Callable[[str], None]) -> None:
25         """Register a callback to receive log lines."""
26         self._log_cb = cb

```

```

27
28 def start(self, world_path: Path) -> None:
29     """Launch Gazebo + ROS 2 for the generated world."""
30     view = self.config.get("view_mode", "3d")
31     launch_file = "sim_2d.launch.py" if view == "2d" else
32     "sim_3d.launch.py"
33
34     launch_path = _SIM_DIR / "ros2" / "launch" / launch_file
35
36     cmd = [
37         "ros2", "launch", str(launch_path),
38         f"world_file:={world_path}",
39         f"scenario:={self.config.get('scenario', 'warehouse')}",
40         f"num_robots:={self.config.get('num_robots', 4)}",
41         f"headless:={'true' if self.config.get('headless') else
42         'false'}",
43         f"rtf:={self.config.get('real_time_factor', 1.0)}",
44     ]
45
46     logger.info(f"Launching: {' '.join(cmd)}")
47
48     try:
49         self._process = subprocess.Popen(
50             cmd,
51             stdout=subprocess.PIPE,
52             stderr=subprocess.STDOUT,
53             text=True,
54             bufsize=1,
55         )
56
57         t = threading.Thread(target=self._stream_log, daemon=True)
58         t.start()
59
60     except Exception as exc:
61         logger.error(f"Failed to start simulation: {exc}")
62         raise
63
64 def stop(self) -> None:
65     """Terminate the simulation process."""

```

```

59         if self._process and self._process.poll() is None:
60             self._process.terminate()
61         try:
62             self._process.wait(timeout=5)
63         except subprocess.TimeoutExpired:
64             self._process.kill()
65
66     def is_running(self) -> bool:
67         """Return True if the simulation process is alive."""
68         return self._process is not None and self._process.poll() is None
69
70     def _stream_log(self) -> None:
71         """Read process stdout and forward to log callback."""
72         if not self._process or not self._process.stdout:
73             return
74         for line in self._process.stdout:
75             logger.debug(line.rstrip())
76             if self._log_cb:
77                 self._log_cb(line.rstrip())

```

Listing C.6: Simulation session manager. Starts the ROS 2 launch subprocess via Popen, exposes `start`, `stop`, and `is_running` for the wizard's launch button.

```

1  """Tracks task allocation state and publishes ROS 2 events."""
2  from __future__ import annotations
3
4  import logging
5  import time
6  from dataclasses import dataclass, field
7  from typing import Any, Dict, List, Optional
8
9  logger = logging.getLogger(__name__)
10

```

```

11
12 @dataclass
13 class TaskRecord:
14     """Immutable description plus mutable lifecycle state for one task."""
15
16     task_id: int
17     description: str
18     position: List[float]
19     status: str = "queued"          # queued / in_progress / done / failed
20     assigned_robot: Optional[int] = None
21     start_time: Optional[float] = None
22     end_time: Optional[float] = None
23
24     @property
25     def duration(self) -> Optional[float]:
26         """Elapsed seconds between assignment and completion, or None."""
27         if self.start_time and self.end_time:
28             return self.end_time - self.start_time
29         return None
30
31
32 class TaskManager:
33     """Central tracker for task allocation during simulation."""
34
35     def __init__(self) -> None:
36         """Initialise with empty task registry."""
37         self._tasks: Dict[int, TaskRecord] = {}
38         self._ros_publisher: Optional[Any] = None
39
40     def init_tasks(self, tasks: List[Dict]) -> None:
41         """Populate task registry from env task list."""
42         self._tasks.clear()
43         for t in tasks:
44             self._tasks[t["id"]] = TaskRecord(

```

```

45         task_id=t["id"],
46         description=t.get("type", f"Task {t['id']}"),
47         position=list(t.get("location", [0.0, 0.0])),
48     )
49
50     def assign(self, task_id: int, robot_id: int) -> None:
51         """Assign task to robot, transition queued→in_progress."""
52         if task_id not in self._tasks:
53             return
54         rec = self._tasks[task_id]
55         rec.assigned_robot = robot_id
56         rec.status         = "in_progress"
57         rec.start_time     = time.time()
58         logger.debug(f"Task {task_id} assigned to robot {robot_id}")
59         self._publish()
60
61     def complete(self, task_id: int) -> None:
62         """Mark task done."""
63         if task_id in self._tasks:
64             self._tasks[task_id].status = "done"
65             self._tasks[task_id].end_time = time.time()
66             self._publish()
67
68     def fail(self, task_id: int) -> None:
69         """Mark task failed."""
70         if task_id in self._tasks:
71             self._tasks[task_id].status = "failed"
72             self._tasks[task_id].end_time = time.time()
73             self._publish()
74
75     def all_tasks(self) -> List[TaskRecord]:
76         """Return snapshot of all task records."""
77         return list(self._tasks.values())
78

```

```

79     def metrics(self) -> Dict[str, Any]:
80         """Compute summary statistics over all tasks."""
81         tasks      = self.all_tasks()
82         done       = [t for t in tasks if t.status == "done"]
83         failed     = [t for t in tasks if t.status == "failed"]
84         durations  = [t.duration for t in done if t.duration is not None]
85         return {
86             "total":      len(tasks),
87             "completed":  len(done),
88             "failed":     len(failed),
89             "avg_duration": float(sum(durations) / len(durations)) if
durations else 0.0,
90         }
91
92     def _publish(self) -> None:
93         """Publish allocation update to ROS 2 if publisher attached."""
94         pass # Filled in by sim_session when ROS is available

```

Listing C.7: TaskRecord dataclass and TaskManager that track task state transitions (queued → in_progress → done/failed) and publish allocation change events to the analytics dashboard.

```

1  """Builds Gazebo SDF .world files from structured world_data dicts."""
2  from __future__ import annotations
3
4  import logging
5  import xml.etree.ElementTree as ET
6  from pathlib import Path
7  from typing import Any, Dict, List
8
9  logger = logging.getLogger(__name__)
10
11
12 def _sub(parent: ET.Element, tag: str, text: str = "") -> ET.Element:

```

```

13     """Create and return a child element, optionally with text content."""
14     el = ET.SubElement(parent, tag)
15     if text:
16         el.text = text
17     return el
18
19
20 class WorldWriter:
21     """Converts a world_data dict into a Gazebo Harmonic SDF world
22     file."""
23
24     def __init__(self, world_data: Dict[str, Any]) -> None:
25         """Store world_data for later serialisation."""
26         self.data = world_data
27
28     def write(self, path: Path) -> None:
29         """Build and write the SDF XML to path."""
30         root = ET.Element("sdf", version="1.9")
31         world = _sub(root, "world")
32         world.set("name", self.data.get("scenario",
33         "aqua_guardian_world"))
34
35         self._add_physics(world)
36         self._add_scene(world)
37         self._add_sun(world)
38         self._add_ground(world)
39         self._add_boundary_walls(world)
40         self._add_static_obstacles(world)
41         self._add_dynamic_obstacles(world)
42         self._add_robots(world)
43         self._add_task_markers(world)
44         self._add_camera_plugin(world)
45
46         tree = ET.ElementTree(root)

```

```

45     ET.indent(tree, space=" ")
46     path.parent.mkdir(parents=True, exist_ok=True)
47     tree.write(str(path), encoding="utf-8", xml_declaration=True)
48     logger.info(f"World written to {path}")
49
50     # -----
51     # Private helpers
52     # -----
53
54     def _add_physics(self, world: ET.Element) -> None:
55         """Add physics block honouring the chosen engine."""
56         physics = _sub(world, "physics")
57         physics.set("name", "default_physics")
58         physics.set("type", self.data.get("physics_engine",
59 "ode").lower())
60         _sub(physics, "max_step_size", "0.001")
61         _sub(physics, "real_time_factor",
62 str(self.data.get("real_time_factor", 1.0)))
63         _sub(physics, "real_time_update_rate", "1000")
64
65     def _add_scene(self, world: ET.Element) -> None:
66         """Configure scene background and shadows."""
67         scene = _sub(world, "scene")
68         ambient = _sub(scene, "ambient")
69         ambient.text = "0.4 0.4 0.4 1"
70         background = _sub(scene, "background")
71         background.text = "0.1 0.1 0.2 1"
72         _sub(scene, "shadows").text = "true"
73
74     def _add_sun(self, world: ET.Element) -> None:
75         """Add a directional sun light."""
76         light = _sub(world, "light")
77         light.set("name", "sun")
78         light.set("type", "directional")

```



```

77     _sub(light, "cast_shadows").text = "true"
78     pose = _sub(light, "pose")
79     pose.text = "0 0 100 0 -0 0"
80     diffuse = _sub(light, "diffuse")
81     diffuse.text = "0.8 0.8 0.8 1"
82     specular = _sub(light, "specular")
83     specular.text = "0.2 0.2 0.2 1"
84     direction = _sub(light, "direction")
85     direction.text = "-0.5 0.1 -0.9"
86
87     def _add_ground(self, world: ET.Element) -> None:
88         """Add a ground plane model."""
89         model = _sub(world, "model")
90         model.set("name", "ground_plane")
91         _sub(model, "static").text = "true"
92         link = _sub(model, "link")
93         link.set("name", "ground_link")
94         coll = _sub(link, "collision")
95         coll.set("name", "ground_collision")
96         geom = _sub(coll, "geometry")
97         plane = _sub(geom, "plane")
98         _sub(plane, "normal").text = "0 0 1"
99         _sub(plane, "size").text = "2000 2000"
100        vis = _sub(link, "visual")
101        vis.set("name", "ground_visual")
102        vgeom = _sub(vis, "geometry")
103        vplane = _sub(vgeom, "plane")
104        _sub(vplane, "normal").text = "0 0 1"
105        _sub(vplane, "size").text = "2000 2000"
106        mat = _sub(vis, "material")
107        _sub(mat, "ambient").text = "0.3 0.35 0.3 1"
108        _sub(mat, "diffuse").text = "0.3 0.35 0.3 1"
109
110    def _add_boundary_walls(self, world: ET.Element) -> None:

```

```

111         """Add four thin box walls at the world boundary."""
112         grid_size = self.data.get("grid_size", [50.0, 50.0])
113         half_w = grid_size[0] / 2.0
114         half_h = grid_size[1] / 2.0
115         wall_thickness = 0.5
116         wall_height = 3.0
117
118         walls = [
119             ("wall_north", 0,          half_h, grid_size[0] +
120             wall_thickness, wall_thickness),
121             ("wall_south", 0,          -half_h, grid_size[0] +
122             wall_thickness, wall_thickness),
123             ("wall_east",  half_w,  0,          wall_thickness,
124             grid_size[1] + wall_thickness),
125             ("wall_west", -half_w,  0,          wall_thickness,
126             grid_size[1] + wall_thickness),
127         ]
128
129         for name, x, y, sx, sy in walls:
130             model = _sub(world, "model")
131             model.set("name", name)
132             _sub(model, "static").text = "true"
133             pose = _sub(model, "pose")
134             pose.text = f"{x} {y} {wall_height/2} 0 0 0"
135             link = _sub(model, "link")
136             link.set("name", "link")
137             coll = _sub(link, "collision")
138             coll.set("name", "coll")
139             geom = _sub(coll, "geometry")
140             box = _sub(geom, "box")
141             _sub(box, "size").text = f"{sx} {sy} {wall_height}"
142             vis = _sub(link, "visual")
143             vis.set("name", "vis")
144             vgeom = _sub(vis, "geometry")
145             vbox = _sub(vgeom, "box")

```

```

141         _sub(vbox, "size").text = f"{sx} {sy} {wall_height}"
142         mat = _sub(vis, "material")
143         _sub(mat, "ambient").text = "0.6 0.6 0.6 1"
144         _sub(mat, "diffuse").text = "0.6 0.6 0.6 1"
145
146     def _add_static_obstacles(self, world: ET.Element) -> None:
147         """Insert one cylinder model per static obstacle."""
148         for idx, obs in enumerate(self.data.get("static_obstacles", [])):
149             model = _sub(world, "model")
150             model.set("name", f"static_obstacle_{idx}")
151             _sub(model, "static").text = "true"
152             pose = _sub(model, "pose")
153             r = obs.get("r", 1.0)
154             pose.text = f"{obs['x']} {obs['y']} {r} 0 0 0"
155             link = _sub(model, "link")
156             link.set("name", "link")
157             for tag in ("collision", "visual"):
158                 el = _sub(link, tag)
159                 el.set("name", tag)
160                 geom = _sub(el, "geometry")
161                 cyl = _sub(geom, "cylinder")
162                 _sub(cyl, "radius").text = str(r)
163                 _sub(cyl, "length").text = str(r * 2)
164                 if tag == "visual":
165                     mat = _sub(el, "material")
166                     _sub(mat, "ambient").text = "0.5 0.5 0.5 1"
167                     _sub(mat, "diffuse").text = "0.5 0.5 0.5 1"
168
169     def _add_dynamic_obstacles(self, world: ET.Element) -> None:
170         """Insert one actor model per dynamic obstacle with a linear
171         trajectory."""
172         for idx, obs in enumerate(self.data.get("dynamic_obstacles", [])):
173             model = _sub(world, "model")
174             model.set("name", f"dynamic_obstacle_{idx}")

```

```

174         _sub(model, "static").text = "false"
175         pose = _sub(model, "pose")
176         r = obs.get("r", 0.5)
177         pose.text = f"{obs['x']} {obs['y']} {r} 0 0 0"
178         link = _sub(model, "link")
179         link.set("name", "link")
180         inertial = _sub(link, "inertial")
181         _sub(inertial, "mass").text = "5.0"
182         for tag in ("collision", "visual"):
183             el = _sub(link, tag)
184             el.set("name", tag)
185             geom = _sub(el, "geometry")
186             box = _sub(geom, "box")
187             _sub(box, "size").text = f"{r*2} {r*2} {r*2}"
188             if tag == "visual":
189                 mat = _sub(el, "material")
190                 _sub(mat, "ambient").text = "1.0 0.5 0.0 1"
191                 _sub(mat, "diffuse").text = "1.0 0.5 0.0 1"
192             # Velocity control plugin
193             plugin = _sub(model, "plugin")
194             plugin.set("filename", "gz-sim-velocity-control-system")
195             plugin.set("name", "gz::sim::systems::VelocityControl")
196             _sub(plugin, "topic").text =
197
198             f"/dynamic_obstacle_{idx}/cmd_vel"
197
198     def _add_robots(self, world: ET.Element) -> None:
199         """Insert a differential drive robot scaffold per robot."""
200         for idx, pos in enumerate(self.data.get("robots", [])):
201             model = _sub(world, "model")
202             model.set("name", f"robot_{idx}")
203             pose = _sub(model, "pose")
204             pose.text = f"{pos[0]} {pos[1]} 0.1 0 0 0"
205             include = _sub(model, "include")
206             _sub(include, "uri").text = "model://robot_base"

```

```

207         _sub(include, "name").text = f"robot_{idx}_base"
208         # Override diff-drive cmd_vel topic per robot
209         plugin = _sub(model, "plugin")
210         plugin.set("filename", "gz-sim-diff-drive-system")
211         plugin.set("name", "gz::sim::systems::DiffDrive")
212         _sub(plugin, "topic").text = f"/robot_{idx}/cmd_vel"
213         _sub(plugin, "odom_topic").text = f"/robot_{idx}/odom"
214
215     def _add_task_markers(self, world: ET.Element) -> None:
216         """Add small glowing cylinder markers for each task location."""
217         for idx, pos in enumerate(self.data.get("tasks", [])):
218             model = _sub(world, "model")
219             model.set("name", f"task_marker_{idx}")
220             _sub(model, "static").text = "true"
221             pose = _sub(model, "pose")
222             pose.text = f"{pos[0]} {pos[1]} 0.25 0 0 0"
223             link = _sub(model, "link")
224             link.set("name", "link")
225             vis = _sub(link, "visual")
226             vis.set("name", "vis")
227             geom = _sub(vis, "geometry")
228             cyl = _sub(geom, "cylinder")
229             _sub(cyl, "radius").text = "0.2"
230             _sub(cyl, "length").text = "0.5"
231             mat = _sub(vis, "material")
232             _sub(mat, "ambient").text = "0.0 1.0 0.6 1"
233             _sub(mat, "diffuse").text = "0.0 1.0 0.6 1"
234             _sub(mat, "emissive").text = "0.0 0.5 0.3 1"
235
236     def _add_camera_plugin(self, world: ET.Element) -> None:
237         """Add a GUI camera plugin AND a sensor camera model that
238         publishes images.
239
240         The GUI camera sets the initial viewpoint when Gazebo is

```

```

displayed.
240     The sensor camera is a separate model with a camera sensor that
publishes
241     images to /overhead_cam/image_raw regardless of whether a display
exists.
242     This is the source used for headless video recording.
243     """
244     view_mode = self.data.get("view_mode", "3d")
245     grid_size = self.data.get("grid_size", [50.0, 50.0])
246     z_height = max(grid_size) * 1.2
247
248     # -- GUI camera (only active when Gazebo window is open) -----
-
249     gui = _sub(world, "gui")
250     plugin = _sub(gui, "plugin")
251     plugin.set("filename", "MinimalScene")
252     plugin.set("name", "MinimalScene")
253     camera = _sub(plugin, "camera_pose")
254     if view_mode == "2d":
255         camera.text = f"0 0 {z_height} 0 1.5708 0"
256     else:
257         diag = (grid_size[0] ** 2 + grid_size[1] ** 2) ** 0.5 * 0.6
258         camera.text = f"{diag} {-diag} {diag*0.8} 0 0.7854 0.7854"
259
260     # -- Overhead sensor camera (always active, publishes to ROS
topic) --
261     # Placed directly above the world centre, pointing straight down.
262     # ros_gz_bridge maps /overhead_cam/image → sensor_msgs/msg/Image.
263     cam_model = _sub(world, "model")
264     cam_model.set("name", "overhead_camera")
265     _sub(cam_model, "static").text = "true"
266     cam_pose = _sub(cam_model, "pose")
267     cam_pose.text = f"0 0 {z_height:.1f} 0 1.5708 0"
268

```

```

269     cam_link = _sub(cam_model, "link")
270     cam_link.set("name", "camera_link")
271
272     sensor = _sub(cam_link, "sensor")
273     sensor.set("name", "overhead_cam")
274     sensor.set("type", "camera")
275     _sub(sensor, "always_on").text = "true"
276     _sub(sensor, "update_rate").text = "30"
277     _sub(sensor, "visualize").text = "false"
278
279     cam_el = _sub(sensor, "camera")
280     _sub(cam_el, "horizontal_fov").text = "1.5708" # 90°
281     image_el = _sub(cam_el, "image")
282     _sub(image_el, "width").text = "1920"
283     _sub(image_el, "height").text = "1080"
284     _sub(image_el, "format").text = "R8G8B8"
285     clip_el = _sub(cam_el, "clip")
286     _sub(clip_el, "near").text = "0.1"
287     _sub(clip_el, "far").text = str(z_height * 2)
288
289     # Gazebo camera plugin that bridges to ROS 2 via ros_gz_bridge
290     cam_plugin = _sub(sensor, "plugin")
291     cam_plugin.set("filename", "gz-sim-sensors-system")
292     cam_plugin.set("name", "gz::sim::systems::Sensors")
293     _sub(cam_plugin, "render_engine").text = "ogre2"

```

Listing C.8: Gazebo Harmonic SDF world writer. Converts a `world_data` dict into valid SDF 1.9 XML covering physics, scene, lighting, boundary walls, static and dynamic obstacles, robot differential-drive scaffolds, task markers, and an overhead camera sensor that publishes to `/overhead_cam/image_raw` for headless recording.

Package: ros2.nodes

Listing C.9: Package init for the ros2 package.

Listing C.10: Package init for the ros2.nodes sub-package.

```
1  """ROS 2 node that publishes simulation metrics and task allocation
   events."""
2  from __future__ import annotations
3
4  import json
5  import logging
6  import sys
7  from typing import Any, Dict
8
9  logger = logging.getLogger(__name__)
10
11  try:
12      import rclpy
13      from rclpy.node import Node
14      from std_msgs.msg import String
15      HAS_ROS = True
16  except ImportError:
17      HAS_ROS = False
18      logger.warning("rclpy not available --- analytics_publisher will not
   function")
19
20
21  if HAS_ROS:
22      class AnalyticsPublisher(Node):
23          """Publishes /sim/metrics at 1 Hz and /sim/task_allocation
   event-driven."""
```



```

24
25     def __init__(self) -> None:
26         """Initialise publishers and 1 Hz metrics timer."""
27         super().__init__("analytics_publisher")
28         self._pub_metrics = self.create_publisher(
29             String, "/sim/metrics", 10
30         )
31         self._pub_allocation = self.create_publisher(
32             String, "/sim/task_allocation", 10
33         )
34         self._pub_event_log = self.create_publisher(
35             String, "/sim/event_log", 10
36         )
37         self._timer = self.create_timer(1.0, self._publish_metrics)
38         self._metrics: Dict[str, Any] = {
39             "total": 0,
40             "completed": 0,
41             "failed": 0,
42             "avg_duration": 0.0,
43             "collision_events": 0,
44             "idle_pct": 0.0,
45             "sim_time": 0.0,
46             "rtf": 1.0,
47         }
48
49     def update_metrics(self, metrics: Dict[str, Any]) -> None:
50         """Merge new metric values into internal state."""
51         self._metrics.update(metrics)
52
53     def publish_allocation_event(self, event: Dict[str, Any]) -> None:
54         """Immediately publish a task allocation change event."""
55         msg = String()
56         msg.data = json.dumps(event)
57         self._pub_allocation.publish(msg)

```

```

58
59     def publish_event_log(self, text: str) -> None:
60         """Publish a single log line to /sim/event_log."""
61         msg = String()
62         msg.data = text
63         self._pub_event_log.publish(msg)
64
65     def _publish_metrics(self) -> None:
66         """Publish current metrics snapshot at 1 Hz."""
67         msg = String()
68         msg.data = json.dumps(self._metrics)
69         self._pub_metrics.publish(msg)
70
71
72 def main() -> None:
73     """Entry point for ros2 run."""
74     if not HAS_ROS:
75         logger.error("rclpy not installed --- cannot start
76         analytics_publisher")
77         sys.exit(1)
78     rclpy.init()
79     node = AnalyticsPublisher()
80     try:
81         rclpy.spin(node)
82     finally:
83         node.destroy_node()
84         rclpy.shutdown()
85
86 if __name__ == "__main__":
87     main()

```

Listing C.11: Analytics publisher node. Publishes a JSON metrics snapshot to `/sim/metrics` at 1 Hz and pushes task allocation change events to `/sim/task_allocation` immediately on each state transition.

```
1  """ROS 2 node that publishes per-robot status at 5 Hz."""
2  from __future__ import annotations
3
4  import json
5  import logging
6  import sys
7
8  logger = logging.getLogger(__name__)
9
10 try:
11     import rclpy
12     from rclpy.node import Node
13     from std_msgs.msg import String
14     HAS_ROS = True
15 except ImportError:
16     HAS_ROS = False
17     logger.warning("rclpy not available --- robot_state_node will not
18         function")
19
20 if HAS_ROS:
21     class RobotStateNode(Node):
22         """Publishes /sim/robot_status at 5 Hz with current robot
23             states."""
24
25         def __init__(self) -> None:
26             """Initialise publisher and 5 Hz timer."""
27             super().__init__("robot_state_node")
```

```

27         self._pub_status = self.create_publisher(
28             String, "/sim/robot_status", 10
29         )
30         self._timer = self.create_timer(0.2, self._tick) # 5 Hz
31         self._robot_states: dict = {}
32
33     def update_robot(self, robot_id: int, data: dict) -> None:
34         """Update internal state for a single robot."""
35         self._robot_states[robot_id] = data
36
37     def _tick(self) -> None:
38         """Publish snapshot of all robot states as JSON."""
39         msg = String()
40         msg.data = json.dumps(self._robot_states)
41         self._pub_status.publish(msg)
42
43
44 def main() -> None:
45     """Entry point for ros2 run."""
46     if not HAS_ROS:
47         logger.error("rclpy not installed --- cannot start
robot_state_node")
48         sys.exit(1)
49     rclpy.init()
50     node = RobotStateNode()
51     try:
52         rclpy.spin(node)
53     finally:
54         node.destroy_node()
55         rclpy.shutdown()
56
57
58 if __name__ == "__main__":
59     main()

```

Listing C.12: Robot state node. Reads per-robot odometry, battery, and task assignment and publishes a JSON status dict to `/sim/robot_status` at 5 Hz.

```
1  """Main ROS 2 bridge node: forwards Gazebo model states to sim topics."""
2  from __future__ import annotations
3
4  import logging
5  import sys
6
7  logger = logging.getLogger(__name__)
8
9  try:
10     import rclpy
11     from rclpy.node import Node
12     from std_msgs.msg import String
13     HAS_ROS = True
14 except ImportError:
15     HAS_ROS = False
16     logger.warning("rclpy not available")
17
18
19 if HAS_ROS:
20     class SimBridgeNode(Node):
21         """Bridges Gazebo model states → /sim/* topics."""
22
23         def __init__(self) -> None:
24             """Create publisher and 10 Hz timer."""
25             super().__init__("sim_bridge_node")
26             self._pub_states = self.create_publisher(String,
27                 "/sim/model_states", 10)
28             self._timer = self.create_timer(0.1, self._tick) # 10 Hz
```

```

28
29     def _tick(self) -> None:
30         """Publish current model states snapshot."""
31         msg = String()
32         msg.data = "{}"    # Filled by actual Gazebo bridge in
                             production
33         self._pub_states.publish(msg)
34
35
36 def main() -> None:
37     """Entry point for ros2 run."""
38     if not HAS_ROS:
39         logger.error("rclpy not installed --- cannot start
sim_bridge_node")
40         sys.exit(1)
41     rclpy.init()
42     node = SimBridgeNode()
43     try:
44         rclpy.spin(node)
45     finally:
46         node.destroy_node()
47         rclpy.shutdown()
48
49
50 if __name__ == "__main__":
51     main()

```

Listing C.13: Simulation bridge node. Subscribes to Gazebo model pose topics and re-publishes the combined world state on `/sim/model_states` at 10 Hz, giving the analytics dashboard a single normalised feed regardless of the Gazebo topic layout.

Package: ros2.launch

Listing C.14: Package init for the ros2.launch sub-package.

```
1  """Shared launch: sim_bridge_node, analytics_publisher,
    robot_state_node."""
2  from __future__ import annotations
3
4  from pathlib import Path
5
6  try:
7      from launch import LaunchDescription
8      from launch_ros.actions import Node
9      HAS_LAUNCH = True
10 except ImportError:
11     HAS_LAUNCH = False
12
13 _SIM_DIR    = Path(__file__).resolve().parent.parent.parent
14 _PARAMS_CFG = str(_SIM_DIR / "ros2" / "config" / "ros2_params.yaml")
15
16
17 def generate_launch_description() -> "LaunchDescription":
18     """Launch the three shared simulation support nodes."""
19     if not HAS_LAUNCH:
20         raise RuntimeError("ros2 launch packages not available")
21
22     sim_bridge = Node(
23         package="aqua_guardian_sim",
24         executable="sim_bridge_node",
25         name="sim_bridge_node",
26         output="screen",
27         parameters=[_PARAMS_CFG],
28     )
```

```

29
30     analytics = Node(
31         package="aqua_guardian_sim",
32         executable="analytics_publisher",
33         name="analytics_publisher",
34         output="screen",
35         parameters=[_PARAMS_CFG],
36     )
37
38     robot_state = Node(
39         package="aqua_guardian_sim",
40         executable="robot_state_node",
41         name="robot_state_node",
42         output="screen",
43         parameters=[_PARAMS_CFG],
44     )
45
46     return LaunchDescription([sim_bridge, analytics, robot_state])

```

Listing C.15: Shared launch description included by every view-mode launch file. Starts `sim_bridge_node`, `analytics_publisher`, and `robot_state_node` with the shared `ros2_params.yaml` configuration.

```

1  """ROS 2 launch file for the AQUA Guardian 2-D top-down simulation
   view."""
2  from __future__ import annotations
3
4  import os
5  from pathlib import Path
6
7  try:
8      from launch import LaunchDescription
9      from launch.actions import (
10         DeclareLaunchArgument,

```



```

11         ExecuteProcess,
12         IncludeLaunchDescription,
13         LogInfo,
14     )
15     from launch.conditions import IfCondition
16     from launch.launch_description_sources import
17         PythonLaunchDescriptionSource
18     from launch.substitutions import LaunchConfiguration, PythonExpression
19     from launch_ros.actions import Node
20     HAS_LAUNCH = True
21 except ImportError:
22     HAS_LAUNCH = False
23
24 _SIM_DIR = Path(__file__).resolve().parent.parent.parent
25 _RVIZ_CFG = str(_SIM_DIR / "ros2" / "config" / "rviz2_2d.rviz")
26 _PARAMS_CFG = str(_SIM_DIR / "ros2" / "config" / "ros2_params.yaml")
27
28 def generate_launch_description() -> "LaunchDescription":
29     """Build the 2-D simulation launch description."""
30     if not HAS_LAUNCH:
31         raise RuntimeError("ros2 launch packages not available")
32
33     # ---- Declared arguments
34     -----
35     world_file_arg = DeclareLaunchArgument(
36         "world_file", default_value="", description="Path to generated
37         .world SDF file"
38     )
39     scenario_arg = DeclareLaunchArgument(
40         "scenario", default_value="warehouse", description="Scenario ID"
41     )
42     num_robots_arg = DeclareLaunchArgument(
43         "num_robots", default_value="4", description="Number of robots"

```

```

42     )
43     headless_arg = DeclareLaunchArgument(
44         "headless", default_value="false", description="Run Gazebo
         headless"
45     )
46     rtf_arg = DeclareLaunchArgument(
47         "rtf", default_value="1.0", description="Real-time factor"
48     )
49
50     world_file = LaunchConfiguration("world_file")
51     headless = LaunchConfiguration("headless")
52     scenario = LaunchConfiguration("scenario")
53     num_robots = LaunchConfiguration("num_robots")
54
55     # ---- Gazebo Sim
56     -----
57     gz_args = PythonExpression([
58         "'-r --headless-rendering ' if '", headless, "' == 'true' else
59         '-r '",
60         "+ str('"', world_file, '"')"
61     ])
62     gz_sim = ExecuteProcess(
63         cmd=["gz", "sim", gz_args],
64         output="screen",
65     )
66
67     # ---- ros_gz_bridge
68     -----
69     try:
70         from launch_ros.actions import Node as RosNode
71         bridge_node = RosNode(
72             package="ros_gz_bridge",
73             executable="parameter_bridge",
74             arguments=[

```

```

72     "/model/robot_0/cmd_vel@geometry_msgs/msg/Twist@gz.msgs.Twist",
73
74     "/model/robot_0/odometry@nav_msgs/msg/Odometry@gz.msgs.Odometry",
75     ],
76     output="screen",
77     parameters=[_PARAMS_CFG],
78 )
79 except Exception:
80     bridge_node = LogInfo(msg="ros_gz_bridge not available --- bridge
81     skipped")
82
83 # ---- Common simulation nodes
84 -----
85 common_launch = IncludeLaunchDescription(
86     PythonLaunchDescriptionSource(
87         str(_SIM_DIR / "ros2" / "launch" / "common_nodes.launch.py")
88     )
89 )
90
91 # ---- RViz2 2-D
92 -----
93 rviz_node = Node(
94     package="rviz2",
95     executable="rviz2",
96     arguments=["-d", _RVIZ_CFG],
97     output="screen",
98     condition=IfCondition(PythonExpression(["'", headless, "' !=
99     'true'"])),
100 )
101
102 return LaunchDescription([
103     world_file_arg,
104     scenario_arg,

```

```

100     num_robots_arg,
101     headless_arg,
102     rtf_arg,
103     gz_sim,
104     bridge_node,
105     common_launch,
106     rviz_node,
107 ])
```

Listing C.16: 2-D top-down launch file. Starts Gazebo Sim, the `ros_gz_bridge` for `cmd_vel` and odometry, the common support nodes, and RViz2 with the 2-D config (suppressed in headless mode).

```

1  """ROS 2 launch file for the AQUA Guardian 3-D perspective simulation
   view."""
2  from __future__ import annotations
3
4  from pathlib import Path
5
6  try:
7      from launch import LaunchDescription
8      from launch.actions import (
9          DeclareLaunchArgument,
10         ExecuteProcess,
11         IncludeLaunchDescription,
12         LogInfo,
13     )
14     from launch.conditions import IfCondition
15     from launch.launch_description_sources import
16         PythonLaunchDescriptionSource
17     from launch.substitutions import LaunchConfiguration, PythonExpression
18     from launch_ros.actions import Node
19     HAS_LAUNCH = True
20 except ImportError:
```

```

20     HAS_LAUNCH = False
21
22     _SIM_DIR     = Path(__file__).resolve().parent.parent.parent
23     _RVIZ_CFG    = str(_SIM_DIR / "ros2" / "config" / "rviz2_3d.rviz")
24     _PARAMS_CFG  = str(_SIM_DIR / "ros2" / "config" / "ros2_params.yaml")
25
26
27 def generate_launch_description() -> "LaunchDescription":
28     """Build the 3-D simulation launch description."""
29     if not HAS_LAUNCH:
30         raise RuntimeError("ros2 launch packages not available")
31
32     # ---- Declared arguments
33     -----
34     world_file_arg = DeclareLaunchArgument(
35         "world_file", default_value="", description="Path to generated
36         .world SDF file"
37     )
38     scenario_arg = DeclareLaunchArgument(
39         "scenario", default_value="warehouse", description="Scenario ID"
40     )
41     num_robots_arg = DeclareLaunchArgument(
42         "num_robots", default_value="4", description="Number of robots"
43     )
44     headless_arg = DeclareLaunchArgument(
45         "headless", default_value="false", description="Run Gazebo
46         headless"
47     )
48     rtf_arg = DeclareLaunchArgument(
49         "rtf", default_value="1.0", description="Real-time factor"
50     )
51
52     world_file = LaunchConfiguration("world_file")
53     headless    = LaunchConfiguration("headless")

```

```

51
52     # ---- Gazebo Sim
53     -----
54     gz_args = PythonExpression([
55         "'-r --headless-rendering ' if '", headless, "' == 'true' else
56         '-r '",
57         "+ str('"', world_file, '"')
58     ])
59     gz_sim = ExecuteProcess(
60         cmd=["gz", "sim", gz_args],
61         output="screen",
62     )
63
64     # ---- ros_gz_bridge
65     -----
66     try:
67         from launch_ros.actions import Node as RosNode
68         bridge_node = RosNode(
69             package="ros_gz_bridge",
70             executable="parameter_bridge",
71             arguments=[
72
73                 "/model/robot_0/cmd_vel@geometry_msgs/msg/Twist@gz.msgs.Twist",
74
75                 "/model/robot_0/odometry@nav_msgs/msg/Odometry@gz.msgs.Odometry",
76
77                 "/world/aqua_guardian_world/pose/info@tf2_msgs/msg/TFMessage@gz.msgs.Pose_V",
78             ],
79             output="screen",
80             parameters=[_PARAMS_CFG],
81         )
82     except Exception:
83         bridge_node = LogInfo(msg="ros_gz_bridge not available --- bridge
84         skipped")

```

```

78
79     # ---- Common simulation nodes
      -----
80     common_launch = IncludeLaunchDescription(
81         PythonLaunchDescriptionSource(
82             str(_SIM_DIR / "ros2" / "launch" / "common_nodes.launch.py")
83         )
84     )
85
86     # ---- RViz2 3-D
      -----
87     rviz_node = Node(
88         package="rviz2",
89         executable="rviz2",
90         arguments=["-d", _RVIZ_CFG],
91         output="screen",
92         condition=IfCondition(PythonExpression(["'", headless, "' !=
'true'"])),
93     )
94
95     return LaunchDescription([
96         world_file_arg,
97         scenario_arg,
98         num_robots_arg,
99         headless_arg,
100        rtf_arg,
101        gz_sim,
102        bridge_node,
103        common_launch,
104        rviz_node,
105    ])

```

Listing C.17: 3-D perspective launch file. Adds the `tf2_msgs/TFMessage` bridge for 6-DOF pose and opens RViz2 with the 3-D config; otherwise identical structure to the 2-D launch file.

Package: analytics

Listing C.18: Package init for the analytics package.

```
1  """Analytics dashboard window: map, task allocation, robot status,
   metrics."""
2  from __future__ import annotations
3
4  import datetime
5  import logging
6  from typing import Any, Dict, List
7
8  from PyQt6.QtCore import Qt, pyqtSlot
9  from PyQt6.QtGui import QColor
10 from PyQt6.QtWidgets import (
11     QMainWindow,
12     QSplitter,
13     QTextEdit,
14     QVBoxLayout,
15     QWidget,
16 )
17
18 from Simulation.analytics.widgets.env_map_widget import EnvMapWidget
19 from Simulation.analytics.widgets.task_allocation_view import
    TaskAllocationView
20 from Simulation.analytics.widgets.robot_status_panel import
```



```

    RobotStatusPanel
21 from Simulation.analytics.widgets.metrics_panel import MetricsPanel
22 from Simulation.analytics.ros_listener import RosListener
23
24 logger = logging.getLogger(__name__)
25
26
27 class AnalyticsDashboard(QMainWindow):
28     """Separate analytics window showing live simulation telemetry."""
29
30     def __init__(self, config: Dict[str, Any], parent: Any = None) ->
    None:
31         """Build the dashboard layout and wire up the ROS listener."""
32         super().__init__(parent)
33         self.config = config
34         self.setWindowTitle("[AQUA Guardian] --- Analytics Dashboard")
35         self.resize(1400, 900)
36
37         # ---- Central widget
    -----
38         central = QWidget()
39         self.setCentralWidget(central)
40         main_layout = QVBoxLayout(central)
41         main_layout.setContentsMargins(4, 4, 4, 4)
42         main_layout.setSpacing(4)
43
44         # ---- Top splitter: map / task table / robot panel
    -----
45         top_splitter = QSplitter(Qt.Orientation.Horizontal)
46
47         self.env_map = EnvMapWidget()
48         self.task_view = TaskAllocationView()
49         self.robot_panel = RobotStatusPanel()
50

```

```

51     top_splitter.addWidget(self.env_map)
52     top_splitter.addWidget(self.task_view)
53     top_splitter.addWidget(self.robot_panel)
54     top_splitter.setStretchFactor(0, 3)
55     top_splitter.setStretchFactor(1, 2)
56     top_splitter.setStretchFactor(2, 1)
57
58     # ---- Bottom splitter: metrics strip / event log
    -----
59     bottom_splitter = QSplitter(Qt.Orientation.Vertical)
60
61     self.metrics_panel = MetricsPanel()
62     self.event_log      = QTextEdit()
63     self.event_log.setReadOnly(True)
64     self.event_log.setMaximumHeight(140)
65     self.event_log.setPlaceholderText("Simulation events will appear
here...")
66
67     bottom_splitter.addWidget(self.metrics_panel)
68     bottom_splitter.addWidget(self.event_log)
69
70     # ---- Outer vertical splitter
    -----
71     outer = QSplitter(Qt.Orientation.Vertical)
72     outer.addWidget(top_splitter)
73     outer.addWidget(bottom_splitter)
74     outer.setStretchFactor(0, 5)
75     outer.setStretchFactor(1, 1)
76
77     main_layout.addWidget(outer)
78
79     # ---- Cross-connect robot panel → map
    -----
80

```

```

81     self.robot_panel.robot_selected.connect(self.env_map.highlight_robot)
82     # ---- ROS listener
83     -----
84     self._listener: RosListener | None = None
85     self._start_ros_listener()
86
87     def _start_ros_listener(self) -> None:
88         """Start the RosListener QThread if rclpy is available."""
89         try:
90             self._listener = RosListener(self)
91
92             self._listener.model_states_received.connect(self._on_model_states)
93
94             self._listener.task_allocation_received.connect(self._on_task_allocation)
95
96             self._listener.robot_status_received.connect(self._on_robot_status)
97
98             self._listener.metrics_received.connect(self.metrics_panel.update_metrics)
99             self._listener.event_log_received.connect(self._on_event_log)
100             self._listener.start()
101         except Exception as exc:
102             logger.warning(f"RosListener not started: {exc}")
103
104     # -----
105     # Slots
106     # -----
107
108     @pyqtSlot(dict)
109     def _on_model_states(self, data: Dict) -> None:
110         """Unpack model states and update map widget."""
111         robots = data.get("robots", [])
112         tasks = data.get("tasks", [])
113         obstacles = data.get("obstacles", [])

```

```

109         self.env_map.update_state(robots, tasks, obstacles)
110
111     @pyqtSlot(dict)
112     def _on_task_allocation(self, data: Dict) -> None:
113         """Forward task list to allocation table."""
114         tasks = data.get("tasks", [])
115         self.task_view.update_allocation(tasks)
116
117     @pyqtSlot(dict)
118     def _on_robot_status(self, data: Dict) -> None:
119         """Distribute per-robot updates to the status panel."""
120         for robot_id_str, robot_data in data.items():
121             try:
122                 rid = int(robot_id_str)
123                 self.robot_panel.update_robot(rid, robot_data)
124             except (ValueError, TypeError) as exc:
125                 logger.debug(f"Skipping robot update: {exc}")
126
127     @pyqtSlot(str)
128     def _on_event_log(self, text: str) -> None:
129         """Append a timestamped line to the event log QTextEdit."""
130         ts = datetime.datetime.now().strftime("%H:%M:%S.%f")[:-3]
131         self.event_log.append(f"[{ts}] {text}")
132
133     def closeEvent(self, event: Any) -> None: # type: ignore[override]
134         """Stop the ROS listener thread on close."""
135         if self._listener and self._listener.isRunning():
136             self._listener.stop()
137         super().closeEvent(event)
138
139
140 def open_dashboard(config: Dict[str, Any]) -> AnalyticsDashboard:
141     """Factory: create and show an AnalyticsDashboard for the given
    config."""

```

```

142     dash = AnalyticsDashboard(config)
143     dash.show()
144     return dash

```

Listing C.19: AnalyticsDashboard QMainWindow. Four-panel layout: 2-D environment map, task allocation table, robot status cards, and metrics strip. Wired to a `RosListener` `QThread` that translates ROS callbacks into Qt signals so the GUI thread is never blocked by `rcipy`.

```

1  """QThread that spins an rcipy node and emits Qt signals for each
   topic."""
2  from __future__ import annotations
3
4  import json
5  import logging
6  from typing import Any, Dict
7
8  logger = logging.getLogger(__name__)
9
10 try:
11     from PyQt6.QtCore import QThread, pyqtSignal
12     HAS_QT = True
13 except ImportError:
14     HAS_QT = False
15     logger.warning("PyQt6 not available --- ros_listener cannot emit Qt
16 signals")
17
18 try:
19     import rcipy
20     from rcipy.node import Node
21     from std_msgs.msg import String
22     HAS_ROS = True
23 except ImportError:
24     HAS_ROS = False

```

```

24     logger.warning("rclpy not available --- ros_listener will not
    subscribe")
25
26
27 if HAS_QT:
28     class RosListener(QThread):
29         """Runs rclpy inside a QThread and emits signals for each sim
    topic."""
30
31         model_states_received = pyqtSignal(dict)
32         task_allocation_received = pyqtSignal(dict)
33         robot_status_received = pyqtSignal(dict)
34         metrics_received = pyqtSignal(dict)
35         event_log_received = pyqtSignal(str)
36
37         def __init__(self, parent: Any = None) -> None:
38             """Initialise with no active node."""
39             super().__init__(parent)
40             self._node: Any = None
41             self._running = False
42
43         def run(self) -> None:
44             """Spin rclpy node in the thread until stop() is called."""
45             if not HAS_ROS:
46                 logger.warning("ROS not available --- RosListener idle")
47                 return
48             rclpy.init()
49             self._node = _ListenerNode(self)
50             self._running = True
51             try:
52                 while self._running and rclpy.ok():
53                     rclpy.spin_once(self._node, timeout_sec=0.1)
54             finally:
55                 if self._node:

```

```

56         self._node.destroy_node()
57         rclpy.shutdown()
58
59     def stop(self) -> None:
60         """Signal the spin loop to exit."""
61         self._running = False
62         self.wait(2000)
63
64
65 if HAS_ROS:
66     class _ListenerNode(Node):
67         """Internal rclpy node that calls back into RosListener."""
68
69         def __init__(self, listener: "RosListener") -> None:
70             """Subscribe to all simulation topics."""
71             super().__init__("aqua_guardian_ros_listener")
72             self._listener = listener
73
74             self.create_subscription(
75                 String, "/sim/model_states", self._on_model_states, 10
76             )
77             self.create_subscription(
78                 String, "/sim/task_allocation",
79                 self._on_task_allocation, 10
80             )
81             self.create_subscription(
82                 String, "/sim/robot_status", self._on_robot_status, 10
83             )
84             self.create_subscription(
85                 String, "/sim/metrics", self._on_metrics, 10
86             )
87             self.create_subscription(
88                 String, "/sim/event_log", self._on_event_log, 10

```

```

89
90     def _parse(self, msg: Any) -> Dict:
91         """Parse a JSON String message into a dict."""
92         try:
93             return json.loads(msg.data)
94         except Exception:
95             return {}
96
97     def _on_model_states(self, msg: Any) -> None:
98         """Forward model states dict to Qt signal."""
99
100     self._listener.model_states_received.emit(self._parse(msg))
101
102     def _on_task_allocation(self, msg: Any) -> None:
103         """Forward task allocation dict to Qt signal."""
104
105     self._listener.task_allocation_received.emit(self._parse(msg))
106
107     def _on_robot_status(self, msg: Any) -> None:
108         """Forward robot status dict to Qt signal."""
109
110     self._listener.robot_status_received.emit(self._parse(msg))
111
112     def _on_metrics(self, msg: Any) -> None:
113         """Forward metrics dict to Qt signal."""
114         self._listener.metrics_received.emit(self._parse(msg))
115
116     def _on_event_log(self, msg: Any) -> None:
117         """Forward raw event log string to Qt signal."""
118         self._listener.event_log_received.emit(msg.data)

```

Listing C.20: RosListener QThread. Spins a private rclpy node inside the thread and emits typed Qt signals (pyqtSignal(dict)) for each of the five /sim/* topics, decoupling the ROS executor from the Qt event loop.

Package: analytics.widgets

Listing C.21: Package init for the analytics.widgets sub-package.

```
1  """2-D environment map widget with pan/zoom and trajectory rendering."""
2  from __future__ import annotations
3
4  import logging
5  import math
6  from collections import deque
7  from typing import Any, Dict, List, Tuple
8
9  from PyQt6.QtCore import Qt, QPointF, QRectF, pyqtSignal, pyqtSlot
10 from PyQt6.QtGui import (
11     QColor,
12     QPainter,
13     QPen,
14     QBrush,
15     QPolygonF,
16     QWheelEvent,
17     QMouseEvent,
18 )
19 from PyQt6.QtWidgets import QWidget
20
21 logger = logging.getLogger(__name__)
```

```

22
23 _ROBOT_COLORS = [
24     QColor("#00BFA5"), QColor("#FF6E40"), QColor("#7C4DFF"),
25     QColor("#FFD740"), QColor("#69F0AE"), QColor("#FF5252"),
26     QColor("#448AFF"), QColor("#E040FB"), QColor("#FFAB40"),
27     QColor("#B9F6CA"),
28 ]
29
30
31 class EnvMapWidget(QWidget):
32     """Paints a real-time top-down map of the simulation environment."""
33
34     robot_highlighted = pyqtSignal(int)
35
36     def __init__(self, parent: Any = None) -> None:
37         """Initialise with empty state and default view transform."""
38         super().__init__(parent)
39         self.setMinimumSize(400, 400)
40         self.world_bounds: Tuple[float, float] = (50.0, 50.0)
41         self.robots: List[Dict] = []
42         self.tasks: List[Dict] = []
43         self.obstacles: List[Dict] = []
44         self._trajectories: Dict[int, deque] = {}
45         self._highlighted_robot: int = -1
46         self._scale: float = 1.0
47         self._offset = QPointF(0.0, 0.0)
48         self._drag_origin: QPointF | None = None
49         self.setMouseTracking(True)
50
51     @pyqtSlot(list, list, list)
52     def update_state(
53         self,
54         robots: List[Dict],
55         tasks: List[Dict],

```

```

56         obstacles: List[Dict],
57     ) -> None:
58         """Receive new simulation state and schedule a repaint."""
59         self.robots    = robots
60         self.tasks     = tasks
61         self.obstacles = obstacles
62         for r in robots:
63             rid = r.get("id", 0)
64             if rid not in self._trajectories:
65                 self._trajectories[rid] = deque(maxlen=200)
66                 x, y = r.get("x", 0.0), r.get("y", 0.0)
67                 self._trajectories[rid].append(QPointF(x, y))
68         self.update()
69
70     @pyqtSlot(int)
71     def highlight_robot(self, robot_id: int) -> None:
72         """Highlight the given robot (called from RobotStatusPanel)."""
73         self._highlighted_robot = robot_id
74         self.update()
75
76     # -----
77     # Coordinate helpers
78     # -----
79
80     def _world_to_widget(self, wx: float, wy: float) -> QPointF:
81         """Map world (m) coordinates to widget pixel coordinates."""
82         cx = self.width() / 2.0 + self._offset.x()
83         cy = self.height() / 2.0 + self._offset.y()
84         px = cx + wx * self._scale * (self.width() / (2.0 *
self.world_bounds[0]))
85         py = cy - wy * self._scale * (self.height() / (2.0 *
self.world_bounds[1]))
86         return QPointF(px, py)
87

```

```

88     def _base_scale(self) -> float:
89         """Pixels-per-metre so the world fills the widget."""
90         sx = self.width() / (2.0 * self.world_bounds[0])
91         sy = self.height() / (2.0 * self.world_bounds[1])
92         return min(sx, sy)
93
94         # -----
95         # Paint
96         # -----
97
98     def paintEvent(self, _event: Any) -> None: # type: ignore[override]
99         """Render world boundary, obstacles, tasks and robots."""
100         p = QPainter(self)
101         p.setRenderHint(QPainter.RenderHint.Antialiasing)
102         p.fillRect(self.rect(), QColor("#1e1e2e"))
103
104         bscale = self._base_scale() * self._scale
105         cx = self.width() / 2.0 + self._offset.x()
106         cy = self.height() / 2.0 + self._offset.y()
107
108         # World boundary
109         bw = self.world_bounds[0] * 2 * bscale
110         bh = self.world_bounds[1] * 2 * bscale
111         p.setPen(QPen(QColor("#00BFA5"), 2))
112         p.setBrush(Qt.BrushStyle.NoBrush)
113         p.drawRect(QRectF(cx - bw / 2, cy - bh / 2, bw, bh))
114
115         # Static obstacles
116         p.setPen(Qt.PenStyle.NoPen)
117         p.setBrush(QBrush(QColor("#555577")))
118         for obs in self.obstacles:
119             if obs.get("dynamic"):
120                 continue
121             pt = self._world_to_widget(obs.get("x", 0), obs.get("y", 0))

```

```

122         r = obs.get("r", 1.0) * bscale
123         p.drawEllipse(pt, r, r)
124
125     # Dynamic obstacles
126     p.setPen(QPen(QColor("#FF6E40"), 2))
127     p.setBrush(Qt.BrushStyle.NoBrush)
128     for obs in self.obstacles:
129         if not obs.get("dynamic"):
130             continue
131         pt = self._world_to_widget(obs.get("x", 0), obs.get("y", 0))
132         r = obs.get("r", 1.0) * bscale
133         p.drawEllipse(pt, r, r)
134
135     # Tasks
136     for t in self.tasks:
137         pt = self._world_to_widget(t.get("x", 0), t.get("y", 0))
138         done = t.get("status") == "done"
139         color = QColor("#888888") if done else QColor("#69F0AE")
140         self._draw_star(p, pt, 6 * self._scale, color)
141
142     # Robot trajectories
143     for rid, traj in self._trajectories.items():
144         if len(traj) < 2:
145             continue
146         color = _ROBOT_COLORS[rid % len(_ROBOT_COLORS)]
147         pts = [self._world_to_widget(pt.x(), pt.y()) for pt in traj]
148         n = len(pts)
149         for i in range(1, n):
150             alpha = int(80 + 175 * i / n)
151             pen = QPen(QColor(color.red(), color.green(),
color.blue(), alpha), 1)
152             p.setPen(pen)
153             p.drawLine(pts[i - 1], pts[i])
154

```

```

155         # Robots
156         for r in self.robots:
157             rid    = r.get("id", 0)
158             pt     = self._world_to_widget(r.get("x", 0), r.get("y", 0))
159             color  = _ROBOT_COLORS[rid % len(_ROBOT_COLORS)]
160             hl     = rid == self._highlighted_robot
161             self._draw_robot(p, pt, r.get("heading", 0.0), color, hl)
162
163         # Legend
164         self._draw_legend(p)
165         p.end()
166
167     def _draw_star(
168         self, p: QPainter, center: QPointF, radius: float, color: QColor
169     ) -> None:
170         """Draw a 5-point star task marker."""
171         p.setPen(Qt.PenStyle.NoPen)
172         p.setBrush(QBrush(color))
173         poly = QPolygonF()
174         for i in range(10):
175             angle = math.pi / 2 + i * math.pi / 5
176             r     = radius if i % 2 == 0 else radius * 0.4
177             poly.append(QPointF(
178                 center.x() + r * math.cos(angle),
179                 center.y() - r * math.sin(angle),
180             ))
181         p.drawPolygon(poly)
182
183     def _draw_robot(
184         self,
185         p: QPainter,
186         center: QPointF,
187         heading: float,
188         color: QColor,

```

```

189         highlighted: bool,
190     ) -> None:
191         """Draw a circle with a heading arrow for one robot."""
192         radius = max(6.0, 8.0 * self._scale)
193         pen_width = 3 if highlighted else 1
194         p.setPen(QPen(color.darker(120), pen_width))
195         p.setBrush(QBrush(color))
196         p.drawEllipse(center, radius, radius)
197         ex = center.x() + (radius + 4) * math.cos(heading)
198         ey = center.y() - (radius + 4) * math.sin(heading)
199         p.setPen(QPen(QColor("#ffffff"), 2))
200         p.drawLine(center, QPointF(ex, ey))
201
202     def _draw_legend(self, p: QPainter) -> None:
203         """Draw a fixed legend in the bottom-left corner."""
204         items = [
205             (QColor("#69F0AE"), "Open task"),
206             (QColor("#888888"), "Done task"),
207             (QColor("#555577"), "Static obstacle"),
208             (QColor("#FF6E40"), "Dynamic obstacle"),
209             (QColor("#00BFA5"), "Robot"),
210         ]
211         x, y = 8, self.height() - 8 - len(items) * 18
212         p.setFont(self.font())
213         for color, label in items:
214             p.setPen(Qt.PenStyle.NoPen)
215             p.setBrush(QBrush(color))
216             p.drawRect(x, y, 12, 12)
217             p.setPen(QPen(QColor("#e0e0e0")))
218             p.drawText(x + 16, y + 11, label)
219             y += 18
220
221         # -----
222         # Mouse interaction (zoom + pan)

```

```

223 # -----
224
225 def wheelEvent(self, event: QWheelEvent) -> None:
226     """Zoom in/out centred on the cursor."""
227     factor = 1.15 if event.angleDelta().y() > 0 else 1 / 1.15
228     self._scale = max(0.2, min(10.0, self._scale * factor))
229     self.update()
230
231 def mousePressEvent(self, event: QMouseEvent) -> None:
232     """Begin pan drag."""
233     if event.button() == Qt.MouseButton.LeftButton:
234         self._drag_origin = event.position()
235
236 def mouseMoveEvent(self, event: QMouseEvent) -> None:
237     """Pan the view while dragging."""
238     if self._drag_origin is not None:
239         delta = event.position() - self._drag_origin
240         self._offset += delta
241         self._drag_origin = event.position()
242         self.update()
243
244 def mouseReleaseEvent(self, event: QMouseEvent) -> None:
245     """End pan drag."""
246     if event.button() == Qt.MouseButton.LeftButton:
247         self._drag_origin = None

```

Listing C.22: EnvMapWidget — custom QWidget that renders a real-time top-down map with pan/zoom, per-robot colour-coded trajectory trails (200-step deque), star task markers, and circle/arrow robot glyphs. A 10-colour palette disambiguates up to ten robots.

```

1 """Horizontal scrollable strip of metric summary cards."""
2 from __future__ import annotations
3

```



```

4 import logging
5 from typing import Any, Dict
6
7 from PyQt6.QtCore import Qt, pyqtSlot
8 from PyQt6.QtGui import QColor
9 from PyQt6.QtWidgets import (
10     QFrame,
11     QHBoxLayout,
12     QLabel,
13     QScrollArea,
14     QSizePolicy,
15     QVBoxLayout,
16     QWidget,
17 )
18
19 logger = logging.getLogger(__name__)
20
21
22 class MetricCard(QFrame):
23     """Single metric display tile."""
24
25     def __init__(self, title: str, unit: str = "", parent: Any = None) ->
26         None:
27         """Create a card showing title and a large value label."""
28         super().__init__(parent)
29         self.setObjectName("MetricCard")
30         self.setStyleSheet(
31             "QFrame#MetricCard { background: #2a2a3e; border-radius: 8px;
32
33             "border: 1px solid #3a3a5e; }"
34         )
35         self.setFixedWidth(140)
36         self.setSizePolicy(QSizePolicy.Policy.Fixed,
37             QSizePolicy.Policy.Expanding)

```

```

35
36     layout = QVBoxLayout(self)
37     layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
38
39     self._value_label = QLabel("---")
40     self._value_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
41     self._value_label.setStyleSheet(
42         "font-size: 22px; font-weight: bold; color: #00BFA5;"
43     )
44
45     title_label = QLabel(title)
46     title_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
47     title_label.setStyleSheet("font-size: 11px; color: #aaaaaa;")
48     title_label.setWordWrap(True)
49
50     layout.addWidget(self._value_label)
51     layout.addWidget(title_label)
52     if unit:
53         unit_label = QLabel(unit)
54         unit_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
55         unit_label.setStyleSheet("font-size: 10px; color: #777777;")
56         layout.addWidget(unit_label)
57
58     def set_value(self, value: str) -> None:
59         """Update the displayed value."""
60         self._value_label.setText(value)
61
62
63 class MetricsPanel(QWidget):
64     """Horizontally scrollable row of MetricCard widgets updated at 1
65     Hz."""
66
67     def __init__(self, parent: Any = None) -> None:
68         """Build the scroll area and pre-create all metric cards."""

```

```

68         super().__init__(parent)
69         self.setFixedHeight(110)
70
71         outer = QHBoxLayout(self)
72         outer.setContentsMargins(0, 0, 0, 0)
73
74         scroll = QScrollArea()
75         scroll.setWidgetResizable(True)
76
77         scroll.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
78
79         scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAsNeeded)
80
81         container = QWidget()
82         self._layout = QHBoxLayout(container)
83         self._layout.setSpacing(8)
84         self._layout.setContentsMargins(4, 4, 4, 4)
85         self._layout.setAlignment(Qt.AlignmentFlag.AlignLeft)
86
87         self._cards: Dict[str, MetricCard] = {}
88         specs = [
89             ("tasks_done", "Tasks Done", ""),
90             ("tasks_failed", "Tasks Failed", ""),
91             ("avg_duration", "Avg Completion", "s"),
92             ("idle_pct", "Idle %", "%"),
93             ("collision_events", "Collisions", ""),
94             ("sim_time", "Sim Time", "s"),
95             ("rtf", "Real-Time Factor", "x"),
96         ]
97         for key, title, unit in specs:
98             card = MetricCard(title, unit, container)
99             self._cards[key] = card
100             self._layout.addWidget(card)

```

```

100         self._layout.addStretch()
101         scroll.setWidget(container)
102         outer.addWidget(scroll)
103
104     @pyqtSlot(dict)
105     def update_metrics(self, data: Dict) -> None:
106         """Refresh card values from a metrics dict."""
107         total      = data.get("total", 0)
108         completed = data.get("completed", 0)
109         self._cards["tasks_done"].set_value(f"{completed}/{total}")
110         self._cards["tasks_failed"].set_value(str(data.get("failed", 0)))
111         avg = data.get("avg_duration", 0.0)
112         self._cards["avg_duration"].set_value(f"{avg:.1f}")
113         self._cards["idle_pct"].set_value(f"{data.get('idle_pct',
114         0.0):.1f}")
115
116         self._cards["collision_events"].set_value(str(data.get("collision_events",
117         0)))
118
119         self._cards["sim_time"].set_value(f"{data.get('sim_time',
120         0.0):.1f}")
121
122         self._cards["rtf"].set_value(f"{data.get('rtf', 1.0):.2f}")

```

Listing C.23: MetricsPanel — horizontally scrollable strip of MetricCard tiles showing tasks done/failed, average completion time, idle percentage, collision count, simulation time, and real-time factor. Updated at 1 Hz from /sim/metrics.

```

1  """Scrollable panel of per-robot status cards."""
2  from __future__ import annotations
3
4  import logging
5  from typing import Any, Dict
6
7  from PyQt6.QtCore import Qt, pyqtSignal, pyqtSlot

```

```

8 from PyQt6.QtGui import QColor
9 from PyQt6.QtWidgets import (
10     QFrame,
11     QHBoxLayout,
12     QLabel,
13     QProgressBar,
14     QScrollArea,
15     QSizePolicy,
16     QVBoxLayout,
17     QWidget,
18 )
19
20 logger = logging.getLogger(__name__)
21
22
23 class RobotCard(QFrame):
24     """Compact status card for a single robot."""
25
26     clicked = pyqtSignal(int)
27
28     def __init__(self, robot_id: int, parent: Any = None) -> None:
29         """Build card layout for the given robot."""
30         super().__init__(parent)
31         self.robot_id = robot_id
32         self setFrameShape(QFrame.Shape.StyledPanel)
33         self.setObjectName("RobotCard")
34         self.setStyleSheet("QFrame#RobotCard { border-radius: 6px;
35         background: #2a2a3e; }")
36
37         self.setCursor(Qt.CursorShape.PointingHandCursor)
38
39         layout = QVBoxLayout(self)
40         layout.setSpacing(2)
41
42         header = QHBoxLayout()

```

```

41     self._id_label      = QLabel(f"Robot {robot_id}")
42     self._id_label.setStyleSheet("font-weight: bold; color: #00BFA5;")
43     self._state_label = QLabel("idle")
44     self._state_label.setAlignment(Qt.AlignmentFlag.AlignRight)
45     header.addWidget(self._id_label)
46     header.addWidget(self._state_label)
47     layout.addLayout(header)
48
49     self._pos_label = QLabel("pos: (0.00, 0.00)")
50     self._pos_label.setStyleSheet("color: #aaaaaa; font-size: 11px;")
51     layout.addWidget(self._pos_label)
52
53     self._task_label = QLabel("task: --- | next: ---")
54     self._task_label.setStyleSheet("color: #aaaaaa; font-size: 11px;")
55     layout.addWidget(self._task_label)
56
57     stats = QHBoxLayout()
58     self._done_label      = QLabel("done: 0")
59     self._distance_label = QLabel("dist: 0.0 m")
60     stats.addWidget(self._done_label)
61     stats.addWidget(self._distance_label)
62     layout.addLayout(stats)
63
64     self._battery = QProgressBar()
65     self._battery.setRange(0, 100)
66     self._battery.setValue(100)
67     self._battery.setTextVisible(True)
68     self._battery.setFormat("Battery: %p%")
69     self._battery.setFixedHeight(12)
70     self._battery.setStyleSheet(
71         "QProgressBar { border: 1px solid #3a3a5e; border-radius:
3px; background: #1a1a2e; }"
72         "QProgressBar::chunk { background: #00BFA5; border-radius:
3px; }"

```

```

73         )
74         layout.addWidget(self._battery)
75
76     def update_data(self, data: Dict) -> None:
77         """Refresh all labels from a robot data dict."""
78         x = data.get("x", 0.0)
79         y = data.get("y", 0.0)
80         self._state_label.setText(data.get("state", "idle"))
81         self._pos_label.setText(f"pos: ({x:.2f}, {y:.2f})")
82         self._task_label.setText(
83             f"task: {data.get('current_task', '---')} | next:
{data.get('next_task', '---')}}"
84         )
85         self._done_label.setText(f"done: {data.get('tasks_done', 0)}")
86         self._distance_label.setText(f"dist: {data.get('distance_m',
0.0):.1f} m")
87         self._battery.setValue(int(data.get("battery", 100)))
88
89     def mousePressEvent(self, _event: Any) -> None: # type:
ignore[override]
90         """Emit clicked signal so the map can highlight this robot."""
91         self.clicked.emit(self.robot_id)
92
93
94 class RobotStatusPanel(QWidget):
95     """Scrollable panel showing one RobotCard per active robot."""
96
97     robot_selected = pyqtSignal(int)
98
99     def __init__(self, parent: Any = None) -> None:
100         """Build scroll area with empty card container."""
101         super().__init__(parent)
102         layout = QVBoxLayout(self)
103         layout.setContentsMargins(0, 0, 0, 0)

```

```

104
105     self._scroll = QScrollArea()
106     self._scroll.setWidgetResizable(True)
107
108     self._scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
109
110     self._container = QWidget()
111     self._c_layout = QVBoxLayout(self._container)
112     self._c_layout.setAlignment(Qt.AlignmentFlag.AlignTop)
113     self._c_layout.setSpacing(4)
114     self._scroll.setWidget(self._container)
115     layout.addWidget(self._scroll)
116
117     self._cards: Dict[int, RobotCard] = {}
118
119     @pyqtSlot(int, dict)
120     def update_robot(self, robot_id: int, data: Dict) -> None:
121         """Update or create the card for robot_id."""
122         if robot_id not in self._cards:
123             card = RobotCard(robot_id, self._container)
124             card.clicked.connect(self.robot_selected)
125             self._cards[robot_id] = card
126             self._c_layout.addWidget(card)
127         self._cards[robot_id].update_data(data)

```

Listing C.24: RobotStatusPanel — scrollable column of RobotCard widgets showing per-robot position, state, current and next task, tasks completed, total distance, and battery level. Clicking a card emits robot_selected to cross-highlight the robot on the map widget.

```

1  """Real-time task allocation table with colour coding and search."""
2  from __future__ import annotations
3
4  import logging

```



```

5 from typing import Any, Dict, List
6
7 from PyQt6.QtCore import Qt, QPropertyAnimation, QEasingCurve, pyqtSlot
8 from PyQt6.QtGui import QColor, QBrush
9 from PyQt6.QtWidgets import (
10     QHBoxLayout,
11     QHeaderView,
12     QLabel,
13     QLineEdit,
14     QTableWidgetItem,
15     QTableWidgetItem,
16     QVBoxLayout,
17     QWidget,
18 )
19
20 logger = logging.getLogger(__name__)
21
22 _STATUS_COLORS: Dict[str, str] = {
23     "queued": "#2979FF",
24     "in_progress": "#FFA000",
25     "done": "#00C853",
26     "failed": "#D50000",
27 }
28
29
30 class TaskAllocationView(QWidget):
31     """Table widget showing live task allocation state."""
32
33     COLUMNS = ["Task ID", "Description", "Robot", "Start Time", "Status",
34                 "Duration"]
35
36     def __init__(self, parent: Any = None) -> None:
37         """Build table and search bar."""
38         super().__init__(parent)

```

```

38     layout = QVBoxLayout(self)
39     layout.setContentsMargins(4, 4, 4, 4)
40
41     # Search bar
42     search_row = QHBoxLayout()
43     search_row.addWidget(QLabel("Filter:"))
44     self._search = QLineEdit()
45     self._search.setPlaceholderText("Search tasks...")
46     self._search.textChanged.connect(self._apply_filter)
47     search_row.addWidget(self._search)
48     layout.addLayout(search_row)
49
50     # Table
51     self._table = QTableWidgetItem(0, len(self.COLUMNS))
52     self._table.setHorizontalHeaderLabels(self.COLUMNS)
53     self._table.horizontalHeader().setSectionResizeMode(
54         QHeaderView.ResizeMode.Stretch
55     )
56     self._table.setAlternatingRowColors(True)
57     self._table.setSortingEnabled(True)
58
59     self._table.setEditTriggers(QTableWidgetItem.EditTrigger.NoEditTriggers)
60     layout.addWidget(self._table)
61
62     @pyqtSlot(list)
63     def update_allocation(self, tasks: List[Dict]) -> None:
64         """Refresh table rows from a list of task dicts."""
65         self._table.setSortingEnabled(False)
66         self._table.setRowCount(len(tasks))
67         for row, t in enumerate(tasks):
68             status = t.get("status", "queued")
69             color = QColor(_STATUS_COLORS.get(status, "#e0e0e0"))
70             dur = t.get("duration")
71             dur_text = f"{dur:.2f}s" if dur is not None else "---"

```

```

71         start    = t.get("start_time")
72         start_text = f"{start:.1f}s" if start is not None else "---"
73         values = [
74             str(t.get("task_id", row)),
75             t.get("description", ""),
76             str(t.get("assigned_robot", "---")),
77             start_text,
78             status,
79             dur_text,
80         ]
81         for col, val in enumerate(values):
82             item = QTableWidgetItem(val)
83             item.setBackground(QBrush(color.darker(140)))
84             item.setForeground(QBrush(QColor("#e0e0e0")))
85             self._table.setItem(row, col, item)
86         self._flash_row(row)
87     self._table.setSortingEnabled(True)
88     self._apply_filter(self._search.text())
89
90     def _flash_row(self, row: int) -> None:
91         """Briefly highlight a row to signal a state change."""
92         for col in range(self._table.columnCount()):
93             item = self._table.item(row, col)
94             if item:
95                 item.setBackground(QBrush(QColor("#00BFA5").darker(110)))
96
97     def _apply_filter(self, text: str) -> None:
98         """Show/hide rows based on text filter."""
99         text = text.lower()
100         for row in range(self._table.rowCount()):
101             match = any(
102                 text in (self._table.item(row, col).text().lower() if
103 self._table.item(row, col) else "")

```

```

104         )
105         self._table.setRowHidden(row, not match and bool(text))

```

Listing C.25: TaskAllocationView — sortable QTableWidgetItem with a live text filter. Columns: Task ID, Description, Robot, Start Time, Status, Duration. Row background is colour-coded by status and flashes teal on each state transition.

Package: gui

Listing C.26: Package init for the gui package.

```

1  """QApplication bootstrap: loads theme and instantiates the wizard."""
2  from __future__ import annotations
3
4  import logging
5  import sys
6  from pathlib import Path
7  from typing import Optional
8
9  from PyQt6.QtWidgets import QApplication
10
11  logger = logging.getLogger(__name__)
12
13  _THEME_PATH = Path(__file__).resolve().parent / "styles" / "theme.qss"
14
15
16  class SimApp(QApplication):
17      """Application class that applies the dark theme and opens the
18          wizard."""

```

```

19     def __init__(self, argv: list) -> None:
20         """Initialise Qt application and load the QSS theme."""
21         super().__init__(argv)
22         self.setApplicationName("AQUA Guardian Simulation Launcher")
23         self.setApplicationVersion("1.0")
24         self._load_theme()
25
26     def _load_theme(self) -> None:
27         """Read and apply the dark theme from theme.qss."""
28         if _THEME_PATH.exists():
29             try:
30                 qss = _THEME_PATH.read_text(encoding="utf-8")
31                 self.setStyleSheet(qss)
32                 logger.info(f"Theme loaded from {_THEME_PATH}")
33             except Exception as exc:
34                 logger.warning(f"Could not load theme: {exc}")
35         else:
36             logger.warning(f"theme.qss not found at {_THEME_PATH}")
37
38     def run(self) -> int:
39         """Show the SimWizard and enter the Qt event loop."""
40         from Simulation.gui.pages.wizard import SimWizard
41         self._wizard = SimWizard()
42         self._wizard.show()
43         return self.exec()

```

Listing C.27: SimApp — QApplication subclass that reads gui/styles/theme.qss and applies the dark colour scheme before showing the SimWizard.

Package: gui.pages

Listing C.28: Package init for the gui.pages sub-package.

```
1  """Environment parameters page with mini-map preview."""
2  from __future__ import annotations
3
4  import logging
5  import math
6  from typing import Any, Dict, List, Optional
7
8  from PyQt6.QtCore import Qt, QPointF, QTimer, pyqtSlot
9  from PyQt6.QtGui import (
10      QBrush,
11      QColor,
12      QPainter,
13      QPen,
14      QPolygonF,
15  )
16  from PyQt6.QtWidgets import (
17      QCheckBox,
18      QDoubleSpinBox,
19      QGroupBox,
20      QHBoxLayout,
21      QLabel,
22      QScrollArea,
23      QSlider,
24      QSpinBox,
25      QTableWidget,
26      QTableWidgetItem,
27      QVBoxLayout,
28      QWidget,
29  )
30
31  logger = logging.getLogger(__name__)
```

```

32
33 SCENARIO_DEFAULTS: Dict[str, Dict] = {
34     "warehouse": {
35         "num_robots": 10, "num_robots_min": 1, "num_robots_max": 20,
36         "num_tasks": 20, "num_tasks_min": 2, "num_tasks_max": 30,
37         "num_static_obstacles": 80, "use_dynamic_obstacles": True,
38         "num_dynamic_obstacles": 3, "dynamic_obstacle_speed": 0.3,
39         "use_dependencies": True, "seed": -1, "max_steps": 750,
40         "world_width": 50.0, "world_height": 50.0,
41     },
42     "search_and_rescue": {
43         "num_robots": 8, "num_robots_min": 1, "num_robots_max": 20,
44         "num_tasks": 30, "num_tasks_min": 2, "num_tasks_max": 50,
45         "num_static_obstacles": 44, "use_dynamic_obstacles": True,
46         "num_dynamic_obstacles": 3, "dynamic_obstacle_speed": 0.3,
47         "use_dependencies": True, "seed": -1, "max_steps": 1000,
48         "world_width": 100.0, "world_height": 100.0,
49     },
50     "maritime_security": {
51         "num_robots": 4, "num_robots_min": 1, "num_robots_max": 20,
52         "num_tasks": 20, "num_tasks_min": 2, "num_tasks_max": 30,
53         "num_static_obstacles": 26, "use_dynamic_obstacles": True,
54         "num_dynamic_obstacles": 3, "dynamic_obstacle_speed": 0.3,
55         "use_dependencies": False, "seed": -1, "max_steps": 1200,
56         "world_width": 500.0, "world_height": 500.0,
57     },
58 }
59
60 REWARD_DISPLAY: Dict[str, Dict] = {
61     "warehouse": {"task_completion_reward": 20.0,
62     "step_penalty": -0.001, "collision_penalty": -0.1},
63     "search_and_rescue": {"task_completion_reward": 50.0,
64     "step_penalty": -0.0005, "collision_penalty": -0.05},
65     "maritime_security": {"task_completion_reward": 80.0,

```

```

        "step_penalty": -0.0002, "collision_penalty": -0.03},
64     }
65
66
67     #
        -----
68     # Mini-map preview widget
69     #
        -----
70
71     class MiniMapPreview(QLabel):
72         """Schematic top-down view of the configured environment."""
73
74         def __init__(self, parent: Optional[QWidget] = None) -> None:
75             """Initialise with empty scene data."""
76             super().__init__(parent)
77             self.setMinimumSize(200, 200)
78             self._world_w: float = 50.0
79             self._world_h: float = 50.0
80             self._num_robots: int = 4
81             self._num_tasks: int = 10
82             self._num_static: int = 20
83             self._num_dynamic: int = 3
84
85         def configure(
86             self,
87             world_w: float,
88             world_h: float,
89             num_robots: int,
90             num_tasks: int,
91             num_static: int,
92             num_dynamic: int,
93         ) -> None:
94             """Update scene configuration and schedule repaint."""

```



```

95     self._world_w = max(world_w, 1.0)
96     self._world_h = max(world_h, 1.0)
97     self._num_robots = num_robots
98     self._num_tasks = num_tasks
99     self._num_static = num_static
100    self._num_dynamic = num_dynamic
101    self.update()
102
103    def paintEvent(self, _event: Any) -> None: # type: ignore[override]
104        """Paint schematic using deterministic positions for preview."""
105        p = QPainter(self)
106        p.setRenderHint(QPainter.RenderHint.Antialiasing)
107        p.fillRect(self.rect(), QColor("#1a1a2e"))
108
109        w, h = self.width(), self.height()
110        margin = 12
111        pw = w - 2 * margin
112        ph = h - 2 * margin
113
114        def to_px(wx: float, wy: float) -> QPointF:
115            px = margin + (wx + self._world_w / 2) / self._world_w * pw
116            py = margin + (1 - (wy + self._world_h / 2) / self._world_h)
117            * ph
118            return QPointF(px, py)
119
120        # Boundary
121        p.setPen(QPen(QColor("#00BFA5"), 2))
122        p.setBrush(Qt.BrushStyle.NoBrush)
123        p.drawRect(margin, margin, pw, ph)
124
125        # Static obstacles (deterministic spiral)
126        p.setPen(Qt.PenStyle.NoPen)
127        p.setBrush(QBrush(QColor("#555577")))
128        half_w = self._world_w / 2

```

```

128     half_h = self._world_h / 2
129     for i in range(min(self._num_static, 60)):
130         angle = i * 2.3999
131         radius = 5 + (i % 5) * half_w / 6
132         ox = math.cos(angle) * radius * (half_w / max(half_w, 1))
133         oy = math.sin(angle) * radius * (half_h / max(half_h, 1))
134         ox = max(-half_w + 2, min(half_w - 2, ox))
135         oy = max(-half_h + 2, min(half_h - 2, oy))
136         pt = to_px(ox, oy)
137         p.drawEllipse(pt, 3, 3)
138
139     # Dynamic obstacles
140     p.setPen(QPen(QColor("#FF6E40"), 1))
141     p.setBrush(Qt.BrushStyle.NoBrush)
142     for i in range(self._num_dynamic):
143         angle = i * (2 * math.pi / max(self._num_dynamic, 1))
144         ox = math.cos(angle) * half_w * 0.6
145         oy = math.sin(angle) * half_h * 0.6
146         pt = to_px(ox, oy)
147         p.drawEllipse(pt, 5, 5)
148
149     # Tasks
150     p.setPen(Qt.PenStyle.NoPen)
151     p.setBrush(QBrush(QColor("#69F0AE")))
152     for i in range(min(self._num_tasks, 30)):
153         tx = math.cos(i * 1.618) * half_w * 0.8
154         ty = math.sin(i * 1.618) * half_h * 0.8
155         pt = to_px(tx, ty)
156         p.drawEllipse(pt, 3, 3)
157
158     # Robots
159     p.setPen(QPen(QColor("#00BFA5"), 1))
160     p.setBrush(QBrush(QColor("#00BFA5")))
161     spacing = self._world_w / max(self._num_robots + 1, 2)

```

```

162         for i in range(self._num_robots):
163             rx = -half_w + spacing * (i + 1)
164             ry = -half_h * 0.85
165             pt = to_px(rx, ry)
166             p.drawEllipse(pt, 4, 4)
167
168         p.end()
169
170
171     #
172     -----
173     # Linked slider + spin box helper
174     #
175     -----
176
177     def _make_slider_spinbox(
178         min_val: int, max_val: int, value: int
179     ) -> tuple["QSlider", "QSpinBox"]:
180         """Create a QSlider and QSpinBox that stay in sync."""
181         slider = QSlider(Qt.Orientation.Horizontal)
182         slider.setRange(min_val, max_val)
183         slider.setValue(value)
184
185         spin = QSpinBox()
186         spin.setRange(min_val, max_val)
187         spin.setValue(value)
188
189         slider.valueChanged.connect(spin.setValue)
190         spin.valueChanged.connect(slider.setValue)
191         return slider, spin
192
193     #
194     -----

```

```

193 # Main page
194 #
195 -----
196 class PageEnvParams(QWidget):
197     """Environment configuration page --- robots, tasks, obstacles,
198     rewards, seed."""
199
200     def __init__(self, parent: Optional[QWidget] = None) -> None:
201         """Build scrollable parameter form and mini-map preview."""
202         super().__init__(parent)
203         self._current_scenario = "warehouse"
204
205         outer = QHBoxLayout(self)
206         outer.setContentsMargins(0, 0, 0, 0)
207         outer.setSpacing(0)
208
209         # Left: scrollable controls
210         scroll = QScrollArea()
211         scroll.setWidgetResizable(True)
212
213         scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
214         scroll_content = QWidget()
215         self._form_layout = QVBoxLayout(scroll_content)
216         self._form_layout.setSpacing(10)
217         self._form_layout.setContentsMargins(16, 12, 16, 12)
218         scroll.setWidget(scroll_content)
219         outer.addWidget(scroll, 3)
220
221         # Right: mini-map
222         right_panel = QWidget()
223         right_panel.setFixedWidth(220)
224         right_layout = QVBoxLayout(right_panel)
225         right_layout.setAlignment(Qt.AlignmentFlag.AlignTop)

```

```

224     map_label = QLabel("Preview")
225     map_label.setStyleSheet("color: #00BFA5; font-weight: bold;")
226     self._mini_map = MiniMapPreview()
227     self._mini_map.setFixedSize(200, 200)
228     right_layout.addWidget(map_label)
229     right_layout.addWidget(self._mini_map)
230     right_layout.addStretch()
231     outer.addWidget(right_panel, 0)
232
233     # Debounce timer for preview
234     self._preview_timer = QTimer()
235     self._preview_timer.setSingleShot(True)
236     self._preview_timer.setInterval(200)
237     self._preview_timer.timeout.connect(self._refresh_preview)
238
239     self._build_controls()
240     self.load_scenario_defaults("warehouse")
241
242     def _build_controls(self) -> None:
243         """Create all collapsible QGroupBox parameter sections."""
244         fl = self._form_layout
245
246         heading = QLabel("Environment Parameters")
247         heading.setStyleSheet("font-size: 18px; font-weight: bold; color:
248         #e0e0e0;")
249         fl.addWidget(heading)
250
251         # 1. Robots
252         robots_box = QGroupBox("Robots")
253         rl = QVBoxLayout(robots_box)
254         self._num_robots_slider, self._num_robots_spin =
255         _make_slider_spinbox(1, 20, 10)
256         self._num_robots_spin.valueChanged.connect(self._schedule_preview)
257         rl.addLayout(self._labeled_row("Num robots",

```

```

self._num_robots_slider, self._num_robots_spin))
256
257     row2 = QHBoxLayout()
258     self._robots_min_spin = QSpinBox();
self._robots_min_spin.setRange(1, 20);
self._robots_min_spin.setValue(1)
259     self._robots_max_spin = QSpinBox();
self._robots_max_spin.setRange(1, 20);
self._robots_max_spin.setValue(20)
260     row2.addWidget(QLabel("Min:"));
row2.addWidget(self._robots_min_spin)
261     row2.addWidget(QLabel("Max:"));
row2.addWidget(self._robots_max_spin)
262     row2.addStretch()
263     rl.addLayout(row2)
264     fl.addWidget(robots_box)
265
266     # 2. Tasks
267     tasks_box = QGroupBox("Tasks")
268     tl = QVBoxLayout(tasks_box)
269     self._num_tasks_slider, self._num_tasks_spin =
_make_slider_spinbox(2, 50, 20)
270     self._num_tasks_spin.valueChanged.connect(self._schedule_preview)
271     tl.addLayout(self._labeled_row("Num tasks",
self._num_tasks_slider, self._num_tasks_spin))
272
273     row3 = QHBoxLayout()
274     self._tasks_min_spin = QSpinBox();
self._tasks_min_spin.setRange(2, 50); self._tasks_min_spin.setValue(2)
275     self._tasks_max_spin = QSpinBox();
self._tasks_max_spin.setRange(2, 50);
self._tasks_max_spin.setValue(30)
276     row3.addWidget(QLabel("Min:"));
row3.addWidget(self._tasks_min_spin)

```

```

277         row3.addWidget(QLabel("Max:"));
row3.addWidget(self._tasks_max_spin)
278         row3.addStretch()
279         tl.addLayout(row3)
280
281         self._use_deps_chk = QCheckBox("Use task dependencies")
282         tl.addWidget(self._use_deps_chk)
283         fl.addWidget(tasks_box)
284
285         # 3. Static obstacles
286         static_box = QGroupBox("Static Obstacles")
287         sl = QVBoxLayout(static_box)
288         self._static_slider, self._static_spin = _make_slider_spinbox(0,
100, 80)
289         self._static_spin.valueChanged.connect(self._schedule_preview)
290         sl.addLayout(self._labeled_row("Count", self._static_slider,
self._static_spin))
291         fl.addWidget(static_box)
292
293         # 4. Dynamic obstacles
294         dyn_box = QGroupBox("Dynamic Obstacles")
295         dl = QVBoxLayout(dyn_box)
296         self._use_dyn_chk = QCheckBox("Enable dynamic obstacles")
297         self._use_dyn_chk.setChecked(True)
298         dl.addWidget(self._use_dyn_chk)
299
300         dyn_row = QHBoxLayout()
301         self._dyn_count_spin = QSpinBox();
self._dyn_count_spin.setRange(0, 10); self._dyn_count_spin.setValue(3)
302         self._dyn_count_spin.valueChanged.connect(self._schedule_preview)
303         self._dyn_speed_spin = QDoubleSpinBox()
304         self._dyn_speed_spin.setRange(0.1, 2.0);
self._dyn_speed_spin.setSingleStep(0.1)
305         self._dyn_speed_spin.setValue(0.3);

```

```

self._dyn_speed_spin.setSuffix(" m/s")
306     dyn_row.addWidget(QLabel("Count:"));
dyn_row.addWidget(self._dyn_count_spin)
307     dyn_row.addWidget(QLabel("Speed:"));
dyn_row.addWidget(self._dyn_speed_spin)
308     dyn_row.addStretch()
309     dl.addLayout(dyn_row)
310     fl.addWidget(dyn_box)
311
312     # 5. Environment bounds
313     env_box = QGroupBox("Environment Bounds")
314     el = QVBoxLayout(env_box)
315     bounds_row = QHBoxLayout()
316     self._world_w_spin = QDoubleSpinBox();
self._world_w_spin.setRange(10, 2000);
self._world_w_spin.setValue(50); self._world_w_spin.setSuffix(" m")
317     self._world_h_spin = QDoubleSpinBox();
self._world_h_spin.setRange(10, 2000);
self._world_h_spin.setValue(50); self._world_h_spin.setSuffix(" m")
318     self._world_w_spin.valueChanged.connect(self._schedule_preview)
319     self._world_h_spin.valueChanged.connect(self._schedule_preview)
320     bounds_row.addWidget(QLabel("Width:"));
bounds_row.addWidget(self._world_w_spin)
321     bounds_row.addWidget(QLabel("Height:"));
bounds_row.addWidget(self._world_h_spin)
322     self._max_steps_spin = QSpinBox();
self._max_steps_spin.setRange(100, 10000);
self._max_steps_spin.setValue(750)
323     bounds_row.addWidget(QLabel("Max steps:"));
bounds_row.addWidget(self._max_steps_spin)
324     bounds_row.addStretch()
325     el.addLayout(bounds_row)
326     fl.addWidget(env_box)
327

```



```

328         # 6. Reward display (read-only)
329         reward_box = QGroupBox("Reward / Done Conditions (read-only)")
330         rv = QVBoxLayout(reward_box)
331         self._reward_table = QTableWidgetItem(3, 2)
332         self._reward_table.setHorizontalHeaderLabels(["Parameter",
333 "Value"])
334
335         self._reward_table.horizontalHeader().setStretchLastSection(True)
336
337         self._reward_table.setEditTriggers(QTableWidgetItem.NoEditTriggers)
338
339         self._reward_table.setMaximumHeight(100)
340
341         rv.addWidget(self._reward_table)
342         fl.addWidget(reward_box)
343
344         # 7. Seed
345         seed_box = QGroupBox("Seed & Reproducibility")
346         seed_l = QHBoxLayout(seed_box)
347         self._seed_spin = QSpinBox(); self._seed_spin.setRange(-1,
348 99999); self._seed_spin.setValue(-1)
349
350         self._seed_spin.setSpecialValueText("Random (1)")
351         self._save_world_chk = QCheckBox("Save world SDF")
352         seed_l.addWidget(QLabel("Seed:"));
353         seed_l.addWidget(self._seed_spin)
354
355         seed_l.addWidget(self._save_world_chk)
356         seed_l.addStretch()
357         fl.addWidget(seed_box)
358
359         fl.addStretch()
360
361     @staticmethod
362     def _labeled_row(label: str, slider: QSlider, spin: QSpinBox) ->
363     QHBoxLayout:
364         """Return a QHBoxLayout with a label, slider, and spinbox."""
365         row = QHBoxLayout()
366         lbl = QLabel(label)

```

```

357         lbl.setFixedWidth(110)
358         row.addWidget(lbl)
359         row.addWidget(slider, 1)
360         row.addWidget(spin)
361         return row
362
363     def load_scenario_defaults(self, scenario: str) -> None:
364         """Reset all controls to scenario-specific default values."""
365         self._current_scenario = scenario
366         d = SCENARIO_DEFAULTS.get(scenario,
367                                   SCENARIO_DEFAULTS["warehouse"])
368
369         self._num_robots_spin.setValue(d["num_robots"])
370         self._robots_min_spin.setValue(d["num_robots_min"])
371         self._robots_max_spin.setValue(d["num_robots_max"])
372         self._num_tasks_spin.setValue(d["num_tasks"])
373         self._tasks_min_spin.setValue(d["num_tasks_min"])
374         self._tasks_max_spin.setValue(d["num_tasks_max"])
375         self._static_spin.setValue(d["num_static_obstacles"])
376         self._use_dyn_chk.setChecked(d["use_dynamic_obstacles"])
377         self._dyn_count_spin.setValue(d["num_dynamic_obstacles"])
378         self._dyn_speed_spin.setValue(d["dynamic_obstacle_speed"])
379         self._use_deps_chk.setChecked(d["use_dependencies"])
380         self._seed_spin.setValue(d["seed"])
381         self._max_steps_spin.setValue(d["max_steps"])
382         self._world_w_spin.setValue(d["world_width"])
383         self._world_h_spin.setValue(d["world_height"])
384
385         rewards = REWARD_DISPLAY.get(scenario, {})
386         rows = list(rewards.items())
387         self._reward_table.setRowCount(len(rows))
388         for i, (k, v) in enumerate(rows):
389             self._reward_table.setItem(i, 0, QTableWidgetItem(k))
390             self._reward_table.setItem(i, 1, QTableWidgetItem(str(v)))

```

```

390
391     self._refresh_preview()
392
393 def _schedule_preview(self) -> None:
394     """Debounce preview refresh to 200 ms after last change."""
395     self._preview_timer.start()
396
397 def _refresh_preview(self) -> None:
398     """Update mini-map with current control values."""
399     self._mini_map.configure(
400         world_w=self._world_w_spin.value(),
401         world_h=self._world_h_spin.value(),
402         num_robots=self._num_robots_spin.value(),
403         num_tasks=self._num_tasks_spin.value(),
404         num_static=self._static_spin.value(),
405         num_dynamic=self._dyn_count_spin.value() if
self._use_dyn_chk.isChecked() else 0,
406     )
407
408 def get_config(self) -> Dict:
409     """Collect all parameter values into a config dict."""
410     return {
411         "num_robots": self._num_robots_spin.value(),
412         "num_robots_min": self._robots_min_spin.value(),
413         "num_robots_max": self._robots_max_spin.value(),
414         "num_tasks": self._num_tasks_spin.value(),
415         "num_tasks_min": self._tasks_min_spin.value(),
416         "num_tasks_max": self._tasks_max_spin.value(),
417         "num_static_obstacles": self._static_spin.value(),
418         "use_dynamic_obstacles": self._use_dyn_chk.isChecked(),
419         "num_dynamic_obstacles": self._dyn_count_spin.value(),
420         "dynamic_obstacle_speed": self._dyn_speed_spin.value(),
421         "use_dependencies": self._use_deps_chk.isChecked(),
422         "seed": self._seed_spin.value(),

```

```

423         "max_steps":            self._max_steps_spin.value(),
424         "world_width":          self._world_w_spin.value(),
425         "world_height":         self._world_h_spin.value(),
426         "save_world_sdf":       self._save_world_chk.isChecked(),
427     }

```

Listing C.29: Environment parameters page. Scrollable form with linked slider/spinbox pairs for robot count, task count, obstacle counts, world bounds, seed, and max steps. A 200 ms debounced `MiniMapPreview` widget re-renders a schematic top-down view on each change. Per-scenario defaults are loaded when the wizard enters this page.

```

1  """Model selection page: browse for .pt files, validate, multi-select."""
2  from __future__ import annotations
3
4  import logging
5  import os
6  from datetime import datetime
7  from pathlib import Path
8  from typing import Dict, List, Optional
9
10 from PyQt6.QtCore import Qt, QThread, pyqtSignal
11 from PyQt6.QtGui import QColor
12 from PyQt6.QtWidgets import (
13     QCheckBox,
14     QFileDialog,
15     QGroupBox,
16     QHBoxLayout,
17     QHeaderView,
18     QLabel,
19     QLineEdit,
20     QPushButton,
21     QSizePolicy,

```

```

22     QTableWidget,
23     QTableWidgetItem,
24     QVBoxLayout,
25     QWidget,
26 )
27
28 logger = logging.getLogger(__name__)
29
30 try:
31     import torch
32     HAS_TORCH = True
33 except ImportError:
34     HAS_TORCH = False
35
36 _DEFAULT_SCAN_DIR = Path(__file__).resolve().parent.parent.parent.parent
37     / "models" / "simulation_ready"
38
39 _COLUMNS = ["Select", "File Name", "Type", "Size", "Modified", "Notes"]
40
41 class _ValidatorThread(QThread):
42     """Background thread that validates a list of model paths."""
43
44     result = pyqtSignal(str, bool, str) # path, ok, message
45
46     def __init__(self, paths: List[str], parent: Optional[QWidget] =
47         None) -> None:
48         """Store paths to validate."""
49         super().__init__(parent)
50         self.paths = paths
51
52     def run(self) -> None:
53         """Run torch.jit.load + dummy forward pass for each path."""
54         from Simulation.core.model_loader import validate_model
55         for path in self.paths:

```

```

54         ok, msg = validate_model(path)
55         self.result.emit(path, ok, msg)
56
57
58 class PageModels(QWidget):
59     """Model browser and validator page."""
60
61     models_validated = pyqtSignal(bool)  # emitted when  $\geq 1$  model is
        validated
62
63     def __init__(self, parent: Optional[QWidget] = None) -> None:
64         """Build model browser layout."""
65         super().__init__(parent)
66         self._validated_paths: List[str] = []
67         self._row_paths: Dict[int, str] = {}
68
69         layout = QVBoxLayout(self)
70         layout.setSpacing(12)
71         layout.setContentsMargins(24, 16, 24, 16)
72
73         heading = QLabel("Select AI Models")
74         heading.setStyleSheet("font-size: 20px; font-weight: bold; color:
        #e0e0e0;")
75         layout.addWidget(heading)
76
77         # Section A: scan directory
78         scan_group = QGroupBox("Scan Directory for .pt / .pth Files")
79         scan_layout = QHBoxLayout(scan_group)
80         self._dir_edit = QLineEdit(str(_DEFAULT_SCAN_DIR))
81         browse_btn = QPushButton("Browse...")
82         browse_btn.clicked.connect(self._browse_dir)
83         scan_btn = QPushButton("Scan")
84         scan_btn.clicked.connect(self._scan_dir)
85         scan_layout.addWidget(self._dir_edit, 1)

```

```

86         scan_layout.addWidget(browse_btn)
87         scan_layout.addWidget(scan_btn)
88         layout.addWidget(scan_group)
89
90         # Section B: file table
91         table_group = QGroupBox("Found Models")
92         table_layout = QVBoxLayout(table_group)
93         self._table = QTableWidgetItem(0, len(_COLUMNS))
94         self._table.setHorizontalHeaderLabels(_COLUMNS)
95         self._table.horizontalHeader().setSectionResizeMode(1,
QHeaderView.ResizeMode.Stretch)
96         self._table.horizontalHeader().setSectionResizeMode(5,
QHeaderView.ResizeMode.Stretch)
97         self._table.setAlternatingRowColors(True)
98
99         self._table.setEditTriggers(QTableWidgetItem.EditTrigger.NoEditTriggers)
100         table_layout.addWidget(self._table)
101         layout.addWidget(table_group, 1)
102
103         # Validate button
104         btn_row = QHBoxLayout()
105         self._validate_btn = QPushButton("Validate selected models")
106         self._validate_btn.clicked.connect(self._validate_selected)
107         self._status_label = QLabel("")
108         self._status_label.setStyleSheet("color: #aaaaaa;")
109         btn_row.addWidget(self._validate_btn)
110         btn_row.addWidget(self._status_label, 1)
111         layout.addLayout(btn_row)
112
113         if not HAS_TORCH:
114             warn = QLabel(" torch not installed --- model validation
disabled")
115             warn.setStyleSheet("color: #FFA000;")
116             layout.addWidget(warn)

```

```

116
117     # Initial scan
118     self._scan_dir()
119
120     def _browse_dir(self) -> None:
121         """Open folder browser and update scan path."""
122         directory = QFileDialog.getExistingDirectory(
123             self, "Select model directory", self._dir_edit.text()
124         )
125         if directory:
126             self._dir_edit.setText(directory)
127             self._scan_dir()
128
129     def _scan_dir(self) -> None:
130         """Recursively find .pt and .pth files and populate table."""
131         scan_path = Path(self._dir_edit.text())
132         if not scan_path.exists():
133             self._status_label.setText("Directory not found")
134             return
135
136         files: List[Path] = list(scan_path.rglob("*.pt")) +
list(scan_path.rglob("*.pth"))
137         files.sort(key=lambda p: p.stat().st_mtime, reverse=True)
138
139         self._table.setRowCount(0)
140         self._row_paths.clear()
141
142         for row, fp in enumerate(files):
143             self._table.insertRow(row)
144             self._row_paths[row] = str(fp)
145
146         # Col 0 --- checkbox
147         chk = QCheckBox()
148         chk_widget = QWidget()

```



```

149         chk_layout = QHBoxLayout(chk_widget)
150         chk_layout.addWidget(chk)
151         chk_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
152         chk_layout.setContentsMargins(0, 0, 0, 0)
153         self._table.setCellWidget(row, 0, chk_widget)
154
155         # Col 1 --- filename
156         self._table.setItem(row, 1, QTableWidgetItem(fp.name))
157
158         # Col 2 --- type inference
159         name = fp.stem.lower()
160         suffix = fp.suffix
161         if "diff" in name:
162             model_type = "Diffusion"
163         elif any(x in name for x in ("ppo", "mappo", "qmix", "coma")):
164             model_type = "RL Policy"
165         else:
166             model_type = ".pt" if suffix == ".pt" else ".pth"
167         self._table.setItem(row, 2, QTableWidgetItem(model_type))
168
169         # Col 3 --- size
170         size_kb = fp.stat().st_size / 1024
171         size_str = f"{size_kb:.1f} KB" if size_kb < 1024 else
172         f"{size_kb/1024:.2f} MB"
173         self._table.setItem(row, 3, QTableWidgetItem(size_str))
174
175         # Col 4 --- modified
176         mtime = datetime.fromtimestamp(fp.stat().st_mtime)
177         self._table.setItem(row, 4,
178         QTableWidgetItem(mtime.strftime("%Y-%m-%d %H:%M")))
179
180         # Col 5 --- notes (empty until validated)
181         self._table.setItem(row, 5, QTableWidgetItem(""))

```

```

181         self._status_label.setText(f"{len(files)} file(s) found")
182
183     def _validate_selected(self) -> None:
184         """Validate all checked model files in a background thread."""
185         if not HAS_TORCH:
186             self._status_label.setText("torch not available --- cannot
187 validate")
188             return
189
190         selected_paths: List[str] = []
191         for row in range(self._table.rowCount()):
192             chk_widget = self._table.cellWidget(row, 0)
193             if chk_widget:
194                 chk = chk_widget.findChild(QCheckBox)
195                 if chk and chk.isChecked():
196                     selected_paths.append(self._row_paths[row])
197
198         if not selected_paths:
199             self._status_label.setText("No models selected")
200             return
201
202         self._status_label.setText(f"Validating {len(selected_paths)}
203 model(s)...")
204
205         self._validate_btn.setEnabled(False)
206
207         self._thread = _ValidatorThread(selected_paths, self)
208         self._thread.result.connect(self._on_validation_result)
209         self._thread.finished.connect(self._on_validation_done)
210         self._thread.start()
211
212     def _on_validation_result(self, path: str, ok: bool, message: str) ->
213 None:
214         """Update the Notes cell for the validated model."""
215         for row, rp in self._row_paths.items():

```

```

212         if rp == path:
213             note_item = self._table.item(row, 5)
214             if note_item is None:
215                 note_item = QTableWidgetItem()
216                 self._table.setItem(row, 5, note_item)
217             note_item.setText(message)
218             color = QColor("#00C853") if ok else QColor("#D50000")
219             note_item.setForeground(color)
220             if ok and path not in self._validated_paths:
221                 self._validated_paths.append(path)
222             elif not ok and path in self._validated_paths:
223                 self._validated_paths.remove(path)
224             break
225
226     def _on_validation_done(self) -> None:
227         """Re-enable validate button and report validated count."""
228         self._validate_btn.setEnabled(True)
229         n = len(self._validated_paths)
230         self._status_label.setText(f"{n} model(s) validated " if n else
231         "No valid models")
232         self.models_validated.emit(n > 0)
233
234     def get_config(self) -> Dict:
235         """Return dict with list of validated model paths."""
236         return {"model_paths": list(self._validated_paths)}

```

Listing C.30: Model selection page. Recursively scans a directory for .pt/.pth files, infers model type from filename, and validates each via torch.jit.load in a background QThread so the GUI stays responsive during disk and inference warmup.

```

1  """Scenario selection page with interactive cards."""
2  from __future__ import annotations
3

```

```

4 import logging
5 from typing import Dict, List, Optional
6
7 from PyQt6.QtCore import Qt
8 from PyQt6.QtGui import QColor
9 from PyQt6.QtWidgets import (
10     QFrame,
11     QGridLayout,
12     QHBoxLayout,
13     QLabel,
14     QSizePolicy,
15     QVBoxLayout,
16     QWidget,
17 )
18
19 logger = logging.getLogger(__name__)
20
21 _SCENARIOS = [
22     {
23         "id": "warehouse",
24         "title": "Warehouse Logistics",
25         "subtitle": "AMRs pick and deliver packages in a structured
26         aisle grid",
27         "icon": "",
28         "tags": ["Multi-robot", "Task Dependencies", "Indoor"],
29         "color": "#2979FF",
30     },
31     {
32         "id": "search_and_rescue",
33         "title": "Search & Rescue",
34         "subtitle": "Land robots navigate rubble to locate and assist
35         victims",
36         "icon": "",
37         "tags": ["Multi-robot", "Dynamic Obstacles", "Outdoor"],

```

```

36         "color":          "#FF6D00",
37     },
38     {
39         "id":              "maritime_security",
40         "title":           "Maritime Security",
41         "subtitle":        "Autonomous vessels patrol and intercept threats
at sea",
42         "icon":            "",
43         "tags":             ["Multi-robot", "Dynamic Env", "Maritime Physics"],
44         "color":            "#00B8D4",
45     },
46 ]
47
48
49 class ScenarioCard(QFrame):
50     """Clickable scenario card that highlights on selection."""
51
52     def __init__(
53         self,
54         meta: Dict,
55         on_select: "callable",
56         parent: Optional[QWidget] = None,
57     ) -> None:
58         """Build card layout from scenario metadata dict."""
59         super().__init__(parent)
60         self.scenario_id = meta["id"]
61         self._on_select = on_select
62         self._accent = meta["color"]
63         self._selected = False
64
65         self.setObjectName("ScenarioCard")
66         self.setSizePolicy(QSizePolicy.Policy.Expanding,
QSizePolicy.Policy.Expanding)
67         self.setMinimumSize(220, 200)

```

```

68     self.setCursor(Qt.CursorShape.PointingHandCursor)
69     self._apply_style(False)
70
71     layout = QVBoxLayout(self)
72     layout.setAlignment(Qt.AlignmentFlag.AlignTop)
73     layout.setSpacing(8)
74     layout.setContentsMargins(16, 16, 16, 16)
75
76     icon_label = QLabel(meta["icon"])
77     icon_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
78     icon_label.setStyleSheet("font-size: 36px; background:
transparent;")
79     layout.addWidget(icon_label)
80
81     title = QLabel(meta["title"])
82     title.setAlignment(Qt.AlignmentFlag.AlignCenter)
83     title.setStyleSheet(
84         f"font-size: 15px; font-weight: bold; color: {self._accent};
background: transparent;"
85     )
86     title.setWordWrap(True)
87     layout.addWidget(title)
88
89     sub = QLabel(meta["subtitle"])
90     sub.setAlignment(Qt.AlignmentFlag.AlignCenter)
91     sub.setWordWrap(True)
92     sub.setStyleSheet("color: #aaaaaa; font-size: 12px; background:
transparent;")
93     layout.addWidget(sub)
94
95     layout.addStretch()
96
97     tags_row = QHBoxLayout()
98     tags_row.setAlignment(Qt.AlignmentFlag.AlignCenter)

```

```

99         for tag in meta["tags"]:
100             tag_label = QLabel(tag)
101             tag_label.setStyleSheet(
102                 f"background: {self._accent}22; color: {self._accent}; "
103                 "border-radius: 4px; padding: 2px 6px; font-size: 10px;"
104             )
105             tags_row.addWidget(tag_label)
106         layout.addLayout(tags_row)
107
108     def _apply_style(self, selected: bool) -> None:
109         """Update border colour based on selection state."""
110         border = f"2px solid {self._accent}" if selected else "2px solid
111         #3a3a5e"
112         self.setStyleSheet(
113             f"QFrame#ScenarioCard {{ background: #2a2a3e; border-radius:
114             10px; border: {border}; }}"
115         )
116
117     def set_selected(self, value: bool) -> None:
118         """Visually mark this card as selected or deselected."""
119         self._selected = value
120         self._apply_style(value)
121
122     def mousePressEvent(self, _event: "object") -> None: # type:
123         ignore[override]
124         """Notify parent page when card is clicked."""
125         self._on_select(self.scenario_id)
126
127
128 class PageScenario(QWidget):
129     """Scenario selection page --- click a card to choose the simulation
130     scenario."""
131
132     def __init__(self, parent: Optional[QWidget] = None) -> None:

```

```

129         """Build scenario card grid with warehouse pre-selected."""
130         super().__init__(parent)
131         self._selected: str = "warehouse"
132         self._cards: Dict[str, ScenarioCard] = {}
133
134         layout = QVBoxLayout(self)
135         layout.setSpacing(16)
136         layout.setContentsMargins(40, 24, 40, 24)
137
138         heading = QLabel("Select Simulation Scenario")
139         heading.setStyleSheet("font-size: 20px; font-weight: bold; color:
#e0e0e0;")
140         heading.setAlignment(Qt.AlignmentFlag.AlignCenter)
141         layout.addWidget(heading)
142
143         sub = QLabel("Choose the environment type for this simulation
run.")
144         sub.setAlignment(Qt.AlignmentFlag.AlignCenter)
145         sub.setStyleSheet("color: #aaaaaa;")
146         layout.addWidget(sub)
147
148         grid = QGridLayout()
149         grid.setSpacing(16)
150         for col, meta in enumerate(_SCENARIOS):
151             card = ScenarioCard(meta, self._select_scenario, self)
152             self._cards[meta["id"]] = card
153             grid.addWidget(card, 0, col)
154         layout.addLayout(grid)
155         layout.addStretch()
156
157         # Pre-select default
158         self._select_scenario("warehouse")
159
160     def _select_scenario(self, scenario_id: str) -> None:

```



```

161         """Deselect all cards and highlight the chosen one."""
162         self._selected = scenario_id
163         for sid, card in self._cards.items():
164             card.set_selected(sid == scenario_id)
165         logger.debug(f"Scenario selected: {scenario_id}")
166
167     def get_config(self) -> Dict:
168         """Return the selected scenario identifier."""
169         return {"scenario": self._selected}

```

Listing C.31: Scenario selection page. Three clickable **ScenarioCard** widgets (Warehouse, Search & Rescue, Maritime Security) with coloured borders on selection. Warehouse is pre-selected by default.

```

1  """Simulation configuration page: view mode, timing, recording, ROS 2
   settings."""
2  from __future__ import annotations
3
4  import logging
5  from pathlib import Path
6  from typing import Any, Dict, Optional
7
8  import yaml
9
10 from PyQt6.QtCore import Qt, pyqtSignal
11 from PyQt6.QtWidgets import (
12     QCheckBox,
13     QComboBox,
14     QFileDialog,
15     QGroupBox,
16     QHBoxLayout,
17     QLabel,
18     QLineEdit,
19     QPushButton,

```

```

20     QRadioButton,
21     QScrollArea,
22     QSlider,
23     QSpinBox,
24     QTextEdit,
25     QVBoxLayout,
26     QWidget,
27 )
28
29 logger = logging.getLogger(__name__)
30
31 _SIM_DIR = Path(__file__).resolve().parent.parent.parent
32 _DEFAULT_OUTPUT = str(_SIM_DIR / "logs")
33
34
35 class PageSimConfig(QWidget):
36     """Page 5 --- simulation timing, recording, ROS 2 and launch
37     controls."""
38
39     simulate_requested = pyqtSignal(dict)
40
41     def __init__(self, parent: Optional[QWidget] = None) -> None:
42         """Build all QGroupBox sections and the live YAML summary."""
43         super().__init__(parent)
44         self._full_config: Dict[str, Any] = {}
45
46         outer = QVBoxLayout(self)
47         outer.setContentsMargins(24, 12, 24, 12)
48         outer.setSpacing(10)
49
50         heading = QLabel("Simulation Configuration")
51         heading.setStyleSheet("font-size: 18px; font-weight: bold; color:
#e0e0e0;")
52         outer.addWidget(heading)

```

```

52
53     scroll = QScrollArea()
54     scroll.setWidgetResizable(True)
55     scroll_content = QWidget()
56     form_layout = QVBoxLayout(scroll_content)
57     form_layout.setSpacing(10)
58     form_layout.setContentsMargins(4, 4, 4, 4)
59     scroll.setWidget(scroll_content)
60     outer.addWidget(scroll, 1)
61
62     # 1. View mode
63     view_box = QGroupBox("View Mode")
64     vl = QHBoxLayout(view_box)
65     self._radio_2d = QRadioButton("2-D top-down")
66     self._radio_3d = QRadioButton("3-D perspective")
67     self._radio_3d.setChecked(True)
68     self._radio_2d.toggled.connect(self._update_summary)
69     vl.addWidget(self._radio_2d)
70     vl.addWidget(self._radio_3d)
71     vl.addStretch()
72     form_layout.addWidget(view_box)
73
74     # 2. Timing
75     timing_box = QGroupBox("Simulation Timing")
76     tl = QVBoxLayout(timing_box)
77
78     rtf_row = QHBoxLayout()
79     rtf_label = QLabel("Real-time factor:")
80     rtf_label.setFixedWidth(140)
81     self._rtf_slider = QSlider(Qt.Orientation.Horizontal)
82     self._rtf_slider.setRange(1, 100)    # 0.1x to 10x (×10)
83     self._rtf_slider.setValue(10)
84     self._rtf_slider.setTickInterval(10)
85     self._rtf_label = QLabel("1.0×")

```

```

86         self._rtf_label.setFixedWidth(40)
87         self._rtf_slider.valueChanged.connect(self._on_rtf_change)
88         rtf_row.addWidget(rtf_label)
89         rtf_row.addWidget(self._rtf_slider, 1)
90         rtf_row.addWidget(self._rtf_label)
91         tl.addLayout(rtf_row)
92
93         timing_row2 = QHBoxLayout()
94         self._max_ep_spin = QSpinBox(); self._max_ep_spin.setRange(10,
3600); self._max_ep_spin.setValue(300); self._max_ep_spin.setSuffix("
s")
95         self._step_ms_spin = QSpinBox(); self._step_ms_spin.setRange(10,
500); self._step_ms_spin.setValue(78); self._step_ms_spin.setSuffix("
ms")
96         self._physics_combo = QComboBox()
97         self._physics_combo.addItem("ODE", "Bullet", "DART"])
98         timing_row2.addWidget(QLabel("Max episode:"))
99         timing_row2.addWidget(self._max_ep_spin)
100        timing_row2.addWidget(QLabel("Step size:"))
101        timing_row2.addWidget(self._step_ms_spin)
102        timing_row2.addWidget(QLabel("Physics:"))
103        timing_row2.addWidget(self._physics_combo)
104        timing_row2.addStretch()
105        tl.addLayout(timing_row2)
106        form_layout.addWidget(timing_box)
107
108        # 3. Recording
109        rec_box = QGroupBox("Recording")
110        rl = QVBoxLayout(rec_box)
111        self._record_state_chk = QCheckBox("Record state log")
112        self._save_traj_chk = QCheckBox("Save trajectory CSV")
113        self._capture_video_chk = QCheckBox("Capture video (ffmpeg)")
114        rl.addWidget(self._record_state_chk)
115        rl.addWidget(self._save_traj_chk)

```

```

116         rl.addWidget(self._capture_video_chk)
117
118         out_row = QHBoxLayout()
119         self._output_edit = QLineEdit(_DEFAULT_OUTPUT)
120         self._output_edit.textChanged.connect(self._update_summary)
121         browse_out = QPushButton("Browse...")
122         browse_out.clicked.connect(self._browse_output)
123         out_row.addWidget(QLabel("Output folder:"))
124         out_row.addWidget(self._output_edit, 1)
125         out_row.addWidget(browse_out)
126         rl.addLayout(out_row)
127         form_layout.addWidget(rec_box)
128
129         # 4. ROS 2 / Gazebo Advanced
130         ros_box = QGroupBox("ROS 2 / Gazebo Advanced")
131         ros_l = QVBoxLayout(ros_box)
132         ros_row1 = QHBoxLayout()
133         self._ros_domain_spin = QSpinBox();
134         self._ros_domain_spin.setRange(0, 255);
135         self._ros_domain_spin.setValue(0)
136
137         self._gz_verbosity_combo = QComboBox();
138         self._gz_verbosity_combo.addItem("0");
139         self._gz_verbosity_combo.addItem("1");
140         self._gz_verbosity_combo.addItem("2");
141         self._gz_verbosity_combo.addItem("3");
142         self._gz_verbosity_combo.addItem("4");
143
144         self._gz_verbosity_combo.setCurrentIndex(1)
145         ros_row1.addWidget(QLabel("ROS Domain ID:"))
146         ros_row1.addWidget(self._ros_domain_spin)
147         ros_row1.addWidget(QLabel("Gazebo verbosity:"))
148         ros_row1.addWidget(self._gz_verbosity_combo)
149         ros_row1.addStretch()
150         ros_l.addLayout(ros_row1)
151
152         ros_row2 = QHBoxLayout()
153         self._headless_chk = QCheckBox("Headless mode (no GUI)")
154         self._auto_rviz_chk = QCheckBox("Auto-open RViz2")
155         self._auto_rviz_chk.setChecked(True)

```

```

147     ros_row2.addWidget(self._headless_chk)
148     ros_row2.addWidget(self._auto_rviz_chk)
149     ros_row2.addStretch()
150     ros_l.addLayout(ros_row2)
151     form_layout.addWidget(ros_box)
152
153     # 5. Live YAML summary
154     summary_box = QGroupBox("Config Summary (live YAML)")
155     sl = QVBoxLayout(summary_box)
156     self._summary_edit = QTextEdit()
157     self._summary_edit.setReadOnly(True)
158     self._summary_edit.setMinimumHeight(120)
159     sl.addWidget(self._summary_edit)
160     form_layout.addWidget(summary_box)
161
162     form_layout.addStretch()
163
164     # Connect all change signals to summary refresh
165     for widget in [
166         self._max_ep_spin, self._step_ms_spin, self._ros_domain_spin,
167     ]:
168         widget.valueChanged.connect(self._update_summary)
169     for widget in [
170         self._record_state_chk, self._save_traj_chk,
self._capture_video_chk,
171         self._headless_chk, self._auto_rviz_chk,
172     ]:
173         widget.toggled.connect(self._update_summary)
174
self._physics_combo.currentTextChanged.connect(self._update_summary)
175
self._gz_verbosity_combo.currentTextChanged.connect(self._update_summary)
176     self._rtf_slider.valueChanged.connect(self._update_summary)
177

```

```

178         self._update_summary()
179
180     def _on_rtf_change(self, val: int) -> None:
181         """Format the RTF label from slider value."""
182         self._rtf_label.setText(f"{val/10:.1f}×")
183         self._update_summary()
184
185     def _browse_output(self) -> None:
186         """Open folder picker for output_folder."""
187         directory = QFileDialog.getExistingDirectory(
188             self, "Select output folder", self._output_edit.text()
189         )
190         if directory:
191             self._output_edit.setText(directory)
192
193     def _update_summary(self, *_args: Any) -> None:
194         """Regenerate the YAML summary block from current values."""
195         cfg = self.get_config()
196         merged = {**self._full_config, **cfg}
197         try:
198             text = yaml.dump(merged, default_flow_style=False,
199 sort_keys=False)
200         except Exception as exc:
201             text = f"# YAML error: {exc}"
202             self._summary_edit.setPlainText(text)
203
204     def set_full_config(self, config: Dict[str, Any]) -> None:
205         """Receive the merged config from all previous pages for the
206 summary."""
207         self._full_config = config
208         self._update_summary()
209
210     def get_config(self) -> Dict[str, Any]:
211         """Return all page-5 parameters."""

```

```

210         return {
211             "view_mode":          "2d" if self._radio_2d.isChecked() else
"3d",
212             "real_time_factor":   self._rtf_slider.value() / 10.0,
213             "max_episode_duration": self._max_ep_spin.value(),
214             "step_size_ms":       self._step_ms_spin.value(),
215             "physics_engine":
self._physics_combo.currentText().lower(),
216             "record_state_log":   self._record_state_chk.isChecked(),
217             "save_trajectory_csv": self._save_traj_chk.isChecked(),
218             "capture_video":      self._capture_video_chk.isChecked(),
219             "output_folder":      self._output_edit.text(),
220             "ros_domain_id":      self._ros_domain_spin.value(),
221             "gazebo_verbosity":
int(self._gz_verbosity_combo.currentText()),
222             "headless":          self._headless_chk.isChecked(),
223             "auto_open_rviz":     self._auto_rviz_chk.isChecked(),
224         }

```

Listing C.32: Simulation configuration page (page 5). Controls for view mode (2-D/3-D), real-time factor slider, episode duration, physics engine, recording options (state log, trajectory CSV, ffmpeg video), ROS 2 domain ID, Gazebo verbosity, and headless mode. A live YAML summary regenerates on every widget change.

```

1  """Welcome page: ASCII banner, dependency status indicators, optional
    install."""
2  from __future__ import annotations
3
4  import logging
5  import subprocess
6  from typing import Dict, List, Tuple
7

```



```

8 from PyQt6.QtCore import Qt, QProcess
9 from PyQt6.QtGui import QColor, QFont
10 from PyQt6.QtWidgets import (
11     QFrame,
12     QHBoxLayout,
13     QLabel,
14     QPushButton,
15     QScrollArea,
16     QSizePolicy,
17     QTextEdit,
18     QVBoxLayout,
19     QWidget,
20 )
21
22 logger = logging.getLogger(__name__)
23
24 _ASCII_BANNER = r"""
25     ---  ---  --  --  ---      ---  -  -  -  ---  ---  ---  -  -  -
26   / _ \ / _ \ | \ / | _ \ / _ \| | | | \ | | | _ \| / _ \ | \ | |
27 | ( ) | ( ) | \| | | / | ( | | | | | ' | | | | | ( ) | | \ | |
28 \_ _ \_ _ _ / | | | | | \_ _ \_ _ _ / | | | | | | | | \_ _ \_ _ _ / | | \_ _ \
29 """
30
31 _SCRIPT_PATH = "Simulation/scripts/check_dependencies.sh"
32 _INSTALL_PATH = "Simulation/scripts/install_dependencies.sh"
33
34
35 class _StatusRow(QWidget):
36     """One dependency indicator row: coloured bullet + label + status
37     text."""
38
39     def __init__(self, label: str, parent: QWidget | None = None) -> None:
40         """Build indicator layout."""
41         super().__init__(parent)

```

```

41         layout = QHBoxLayout(self)
42         layout.setContentsMargins(0, 2, 0, 2)
43
44         self._bullet = QLabel("*")
45         self._bullet.setFixedWidth(20)
46         self._bullet.setStyleSheet("color: #555577; font-size: 16px;")
47         self._name = QLabel(label)
48         self._name.setFixedWidth(180)
49         self._status = QLabel("checking...")
50         self._status.setStyleSheet("color: #aaaaaa;")
51
52         layout.addWidget(self._bullet)
53         layout.addWidget(self._name)
54         layout.addWidget(self._status, 1)
55
56     def set_ok(self, detail: str = "OK") -> None:
57         """Mark indicator as passing."""
58         self._bullet.setStyleSheet("color: #00C853; font-size: 16px;")
59         self._status.setText(detail)
60         self._status.setStyleSheet("color: #00C853;")
61
62     def set_warn(self, detail: str) -> None:
63         """Mark indicator as warning."""
64         self._bullet.setStyleSheet("color: #FFA000; font-size: 16px;")
65         self._status.setText(detail)
66         self._status.setStyleSheet("color: #FFA000;")
67
68     def set_fail(self, detail: str = "missing") -> None:
69         """Mark indicator as failing."""
70         self._bullet.setStyleSheet("color: #D50000; font-size: 16px;")
71         self._status.setText(f" {detail}")
72         self._status.setStyleSheet("color: #D50000;")
73
74

```

```

75 class PageWelcome(QWidget):
76     """First wizard page --- project overview and dependency health
       check."""
77
78     _DEP_KEYS: List[Tuple[str, str]] = [
79         ("ROS2_JAZZY",    "ROS 2 Jazzy"),
80         ("GZ_HARMONIC",   "Gazebo Harmonic"),
81         ("GPU_VRAM",       "GPU / VRAM"),
82         ("PYQT6",          "PyQt6"),
83         ("RCLPY",          "rclpy"),
84         ("FFMPEG",         "ffmpeg"),
85     ]
86
87     def __init__(self, parent: QWidget | None = None) -> None:
88         """Build welcome layout."""
89         super().__init__(parent)
90         layout = QVBoxLayout(self)
91         layout.setAlignment(Qt.AlignmentFlag.AlignTop)
92         layout.setSpacing(12)
93         layout.setContentsMargins(40, 20, 40, 20)
94
95         banner = QLabel(_ASCII_BANNER)
96         banner.setFont(QFont("Courier New", 9))
97         banner.setStyleSheet("color: #00BFA5;")
98         banner.setAlignment(Qt.AlignmentFlag.AlignCenter)
99         layout.addWidget(banner)
100
101         h1 = QLabel("AQUA Guardian")
102         h1.setAlignment(Qt.AlignmentFlag.AlignCenter)
103         h1.setStyleSheet("font-size: 24px; font-weight: bold; color:
#e0e0e0;")
104         layout.addWidget(h1)
105
106         h3 = QLabel("Simulation Launcher")

```

```

107     h3.setAlignment(Qt.AlignmentFlag.AlignCenter)
108     h3.setStyleSheet("font-size: 15px; color: #aaaaaa;")
109     layout.addWidget(h3)
110
111     rule = QFrame()
112     rule.setFrameShape(QFrame.Shape.HLine)
113     rule.setStyleSheet("border: 1px solid #3a3a5e;")
114     layout.addWidget(rule)
115
116     body = QLabel(
117         "This wizard configures and launches a full multi-robot
simulation "
118         "using <b>Gazebo Harmonic</b> and <b>ROS 2 Jazzy</b>. "
119         "Trained RL policies are loaded from TorchScript
<code>.pt</code> files "
120         "and executed in real time inside the simulation.\n\n"
121         "Complete each step, then press <b> SIMULATE</b> to launch."
122     )
123     body.setWordWrap(True)
124     body.setStyleSheet("color: #cccccc; line-height: 1.6;")
125     layout.addWidget(body)
126
127     deps_label = QLabel("Dependency Check")
128     deps_label.setStyleSheet("font-size: 14px; font-weight: bold;
color: #00BFA5; margin-top: 8px;")
129     layout.addWidget(deps_label)
130
131     self._status_rows: Dict[str, _StatusRow] = {}
132     for key, label in self._DEP_KEYS:
133         row = _StatusRow(label, self)
134         self._status_rows[key] = row
135         layout.addWidget(row)
136
137     btn_row = QHBoxLayout()

```

```

138     refresh_btn = QPushButton("Refresh checks")
139     refresh_btn.clicked.connect(self.check_deps)
140     btn_row.addWidget(refresh_btn)
141
142     self._install_btn = QPushButton("Install missing dependencies...")
143     self._install_btn.setVisible(False)
144     self._install_btn.clicked.connect(self._run_install)
145     btn_row.addWidget(self._install_btn)
146     btn_row.addStretch()
147     layout.addLayout(btn_row)
148
149     # Hidden until install is triggered by the user
150     self._log_edit = QTextEdit()
151     self._log_edit.setReadOnly(True)
152     self._log_edit.setVisible(False)
153     self._log_edit.setMaximumHeight(150)
154     layout.addWidget(self._log_edit)
155
156     layout.addStretch()
157
158     self.check_deps()
159
160     def check_deps(self) -> None:
161         """Run check_dependencies.sh and parse KEY: STATUS lines."""
162         import sys
163         from pathlib import Path
164
165         script = Path(__file__).resolve().parent.parent.parent.parent /
166         _SCRIPT_PATH
167         if not script.exists():
168             logger.warning(f"check_dependencies.sh not found at {script}")
169             for row in self._status_rows.values():
170                 row.set_warn("script not found")
171         return

```

```

171
172     try:
173         result = subprocess.run(
174             ["bash", str(script)],
175             capture_output=True,
176             text=True,
177             timeout=20,
178         )
179         self._parse_check_output(result.stdout)
180     except FileNotFoundError:
181         # bash not available (Windows without WSL)
182         self._windows_fallback()
183     except subprocess.TimeoutExpired:
184         for row in self._status_rows.values():
185             row.set_warn("timeout")
186
187     def _parse_check_output(self, output: str) -> None:
188         """Parse lines of 'KEY: STATUS' from check_dependencies.sh."""
189         has_missing = False
190         parsed: Dict[str, str] = {}
191         for line in output.splitlines():
192             if ": " in line:
193                 key, _, status = line.partition(": ")
194                 parsed[key.strip()] = status.strip()
195         for key, row in self._status_rows.items():
196             status = parsed.get(key, "")
197             if not status:
198                 row.set_warn("not checked")
199             elif status.startswith("OK"):
200                 row.set_ok(status)
201             elif status.startswith("WARN"):
202                 row.set_warn(status)
203                 has_missing = True
204             elif status.startswith("MISSING") or status.startswith(""):

```

```

205         row.set_fail(status)
206         has_missing = True
207     else:
208         row.set_warn(status)
209     self._install_btn.setVisible(has_missing)
210
211 def _windows_fallback(self) -> None:
212     """On Windows without bash, check Python/PyQt6 directly."""
213     import sys
214     for key, row in self._status_rows.items():
215         if key == "PYQT6":
216             try:
217                 from PyQt6.QtCore import PYQT_VERSION_STR
218                 row.set_ok(f"PyQt6 {PYQT_VERSION_STR}")
219             except ImportError:
220                 row.set_fail("PyQt6 not installed")
221         elif key == "GPU_VRAM":
222             row.set_warn("Cannot check on Windows without bash")
223         else:
224             row.set_warn("bash required for this check")
225
226 def _run_install(self) -> None:
227     """Launch install_dependencies.sh in a QProcess with live
228     output."""
229     from pathlib import Path
230
231     script = Path(__file__).resolve().parent.parent.parent.parent /
232     _INSTALL_PATH
233
234     self._log_edit.setVisible(True)
235     self._log_edit.clear()
236     self._log_edit.append(f"Running: bash {script}\n")
237
238     self._process = QProcess(self)
239     self._process.setProgram("bash")

```

```

237         self._process.setArguments([str(script)])
238
239     self._process.readyReadStandardOutput.connect(self._on_install_stdout)
240
241     self._process.readyReadStandardError.connect(self._on_install_stderr)
242     self._process.finished.connect(self._on_install_finished)
243     self._process.start()
244
245     def _on_install_stdout(self) -> None:
246         """Append stdout from install process to log."""
247         data =
248         self._process.readAllStandardOutput().data().decode(errors="replace")
249         self._log_edit.append(data)
250
251     def _on_install_stderr(self) -> None:
252         """Append stderr from install process to log."""
253         data =
254         self._process.readAllStandardError().data().decode(errors="replace")
255         self._log_edit.append(f"<span
256         style='color:#FF6E40'>{data}</span>")
257
258     def _on_install_finished(self, exit_code: int, _status: int) -> None:
259         """Re-run dependency check after install completes."""
260         self._log_edit.append(f"\n[Installer exited with code
261         {exit_code}]")
262         self.check_deps()
263
264     def get_config(self) -> Dict:
265         """Return empty dict --- welcome page contributes no config."""
266         return {}

```

Listing C.33: Welcome page (page 1). Renders the ASCII banner, project description, and a dependency health check table with colour-coded status rows for ROS 2 Jazzy, Gazebo Harmonic, GPU/VRAM, PyQt6, rclpy, and ffmpeg. Runs `check_dependencies.sh` in a subprocess; falls back to a direct Python import check on Windows without `bash`.

```
1  """Main wizard window: 5-page step flow with animated slide
   transitions."""
2  from __future__ import annotations
3
4  import logging
5  from pathlib import Path
6  from typing import Any, Dict, Optional
7
8  from PyQt6.QtCore import (
9      Qt,
10     QPropertyAnimation,
11     QEasingCurve,
12     QRect,
13     QSize,
14 )
15 from PyQt6.QtGui import QColor
16 from PyQt6.QtWidgets import (
17     QHBoxLayout,
18     QLabel,
19     QMainWindow,
20     QPushButton,
21     QSizePolicy,
22     QStackedWidget,
23     QVBoxLayout,
24     QWidget,
25 )
```

```

26
27 from Simulation.gui.pages.page_welcome    import PageWelcome
28 from Simulation.gui.pages.page_scenario    import PageScenario
29 from Simulation.gui.pages.page_models      import PageModels
30 from Simulation.gui.pages.page_env_params  import PageEnvParams
31 from Simulation.gui.pages.page_sim_config  import PageSimConfig
32
33 logger = logging.getLogger(__name__)
34
35
36 class SimWizard(QMainWindow):
37     """5-step wizard that collects simulation config and launches the
38         sim."""
39
40     _PAGE_TITLES = [
41         "Welcome",
42         "Scenario",
43         "Models",
44         "Environment",
45         "Launch Config",
46     ]
47
48     def __init__(self, parent: Optional[QWidget] = None) -> None:
49         """Build header, stacked pages, and bottom navigation bar."""
50         super().__init__(parent)
51         self.setWindowTitle("[AQUA Guardian] --- Simulation Launcher")
52         self.setMinimumSize(900, 640)
53
54         self.sim_config: Dict[str, Any] = {}
55         self._current_page = 0
56
57         # ---- Root layout
58         -----
59         root = QWidget()

```

```

58     self.setCentralWidget(root)
59     root_layout = QVBoxLayout(root)
60     root_layout.setContentsMargins(0, 0, 0, 0)
61     root_layout.setSpacing(0)
62
63     # ---- Header bar
        -----
64     self._header = QWidget()
65     self._header.setFixedHeight(48)
66     self._header.setStyleSheet("background: #151525; border-bottom:
2px solid #3a3a5e;")
67     header_row = QHBoxLayout(self._header)
68     header_row.setContentsMargins(16, 0, 16, 0)
69
70     self._project_label = QLabel("AQUA Guardian")
71     self._project_label.setStyleSheet("font-size: 15px; font-weight:
bold; color: #00BFA5;")
72     self._step_label = QLabel("Step 1 of 5")
73     self._step_label.setStyleSheet("color: #aaaaaa; font-size: 13px;")
74
75     header_row.addWidget(self._project_label)
76     header_row.addStretch()
77     header_row.addWidget(self._step_label)
78     root_layout.addWidget(self._header)
79
80     # ---- Pages
        -----
81     self._stack = QStackedWidget()
82     self._page_welcome = PageWelcome()
83     self._page_scenario = PageScenario()
84     self._page_models = PageModels()
85     self._page_env = PageEnvParams()
86     self._page_sim_config = PageSimConfig()
87

```

```

88         for page in [
89             self._page_welcome,
90             self._page_scenario,
91             self._page_models,
92             self._page_env,
93             self._page_sim_config,
94         ]:
95             self._stack.addWidget(page)
96
97         root_layout.addWidget(self._stack, 1)
98
99         # ---- Bottom nav bar
100         -----
101         self._nav_bar = QWidget()
102         self._nav_bar.setFixedHeight(52)
103         self._nav_bar.setStyleSheet("background: #151525; border-top: 2px
solid #3a3a5e;")
104         nav_row = QHBoxLayout(self._nav_bar)
105         nav_row.setContentsMargins(16, 0, 16, 0)
106
107         self._back_btn = QPushButton("← Back")
108         self._back_btn.setFixedWidth(110)
109         self._back_btn.clicked.connect(self.go_back)
110
111         self._next_btn = QPushButton("Next →")
112         self._next_btn.setFixedWidth(110)
113         self._next_btn.clicked.connect(self.go_next)
114
115         self._sim_btn = QPushButton("  SIMULATE")
116         self._sim_btn.setFixedWidth(160)
117         self._sim_btn.setStyleSheet(
118             "background: #00BFA5; color: #1e1e2e; font-weight: bold;
font-size: 14px;"
119             "border-radius: 5px; padding: 6px 16px;"

```

```

119         )
120         self._sim_btn.setVisible(False)
121         self._sim_btn.clicked.connect(self._on_simulate)
122
123         nav_row.addWidget(self._back_btn)
124         nav_row.addStretch()
125         nav_row.addWidget(self._next_btn)
126         nav_row.addWidget(self._sim_btn)
127         root_layout.addWidget(self._nav_bar)
128
129         self._update_nav()
130
131         # -----
132         # Navigation
133         # -----
134
135     def go_next(self) -> None:
136         """Advance to the next page with a slide animation."""
137         if self._current_page >= len(self._PAGE_TITLES) - 1:
138             return
139         self._collect_page_config()
140         self._animate_slide(forward=True)
141         self._current_page += 1
142         self._stack.setCurrentIndex(self._current_page)
143         self._on_page_entered()
144         self._update_nav()
145
146     def go_back(self) -> None:
147         """Return to the previous page with a reverse slide animation."""
148         if self._current_page <= 0:
149             return
150         self._animate_slide(forward=False)
151         self._current_page -= 1
152         self._stack.setCurrentIndex(self._current_page)

```

```

153         self._update_nav()
154
155     def _update_nav(self) -> None:
156         """Show/hide and enable/disable nav buttons for current page."""
157         n = len(self._PAGE_TITLES)
158         self._back_btn.setEnabled(self._current_page > 0)
159         is_last = self._current_page == n - 1
160         self._next_btn.setVisible(not is_last)
161         self._sim_btn.setVisible(is_last)
162         self._step_label.setText(f"Step {self._current_page + 1} of {n}
--- {self._PAGE_TITLES[self._current_page]}")
163
164     def _on_page_entered(self) -> None:
165         """Hook called when a new page becomes current."""
166         # When entering env params, reload scenario defaults from chosen
scenario
167         if self._current_page == 3:
168             scenario = self.sim_config.get("scenario", "warehouse")
169             self._page_env.load_scenario_defaults(scenario)
170             # When entering sim config, pass full merged config for YAML
summary
171             if self._current_page == 4:
172                 self._collect_page_config()
173                 self._page_sim_config.set_full_config(dict(self.sim_config))
174
175     def _collect_page_config(self) -> None:
176         """Merge the current page's config into sim_config."""
177         pages = [
178             self._page_welcome,
179             self._page_scenario,
180             self._page_models,
181             self._page_env,
182             self._page_sim_config,
183         ]

```

```

184         page = pages[self._current_page]
185         if hasattr(page, "get_config"):
186             self.sim_config.update(page.get_config())
187
188     def get_config(self) -> Dict[str, Any]:
189         """Return the merged config dict from all pages."""
190         for page in [
191             self._page_welcome,
192             self._page_scenario,
193             self._page_models,
194             self._page_env,
195             self._page_sim_config,
196         ]:
197             if hasattr(page, "get_config"):
198                 self.sim_config.update(page.get_config())
199         return dict(self.sim_config)
200
201     # -----
202     # Slide animation
203     # -----
204
205     def _animate_slide(self, forward: bool) -> None:
206         """Animate the stacked widget geometry for a slide effect."""
207         current_widget = self._stack.currentWidget()
208         if not current_widget:
209             return
210         width = self._stack.width()
211         start_rect = QRect(0, 0, width, self._stack.height())
212         offset = -width if forward else width
213         end_rect = QRect(offset, 0, width, self._stack.height())
214
215         anim = QPropertyAnimation(current_widget, b"geometry")
216         anim.setDuration(150)
217         anim.setEasingCurve(QEasingCurve.Type.OutCubic)

```

```

218         anim.setStartValue(start_rect)
219         anim.setEndValue(end_rect)
220         anim.start()
221         self._anim = anim    # keep reference
222
223     # -----
224     # Simulate
225     # -----
226
227     def _on_simulate(self) -> None:
228         """Collect final config, serialise, build world, and launch."""
229         import yaml
230         from pathlib import Path
231
232         config = self.get_config()
233         _SIM_DIR = Path(__file__).resolve().parent.parent.parent
234         config_path = _SIM_DIR / "configs" / "last_run.yaml"
235         config_path.parent.mkdir(parents=True, exist_ok=True)
236         with config_path.open("w") as fh:
237             yaml.dump(config, fh, default_flow_style=False)
238         logger.info(f"Config saved to {config_path}")
239
240         self._sim_btn.setEnabled(False)
241         self._sim_btn.setText("Launching...")
242
243         try:
244             from Simulation.core.env_builder import build_world
245             world_path = build_world(config)
246
247             from Simulation.core.sim_session import SimSession
248             session = SimSession(config)
249             session.start(world_path)
250
251             from Simulation.analytics.dashboard import open_dashboard

```



```

252         self._dashboard = open_dashboard(config)
253
254         self._sim_btn.setText("  Running")
255     except Exception as exc:
256         logger.error(f"Simulation launch failed: {exc}")
257         self._sim_btn.setText("  SIMULATE")
258         self._sim_btn.setEnabled(True)

```

Listing C.34: SimWizard — 5-page QMainWindow with a QStackedWidget and animated 150 ms slide transitions. Collects merged config from all pages via `get_config()`, serialises to `configs/last_run.yaml`, calls `build_world`, launches `SimSession`, and opens the analytics dashboard on the final **SIMULATE** button press.

Scripts

```

1  """Post-simulation export: trajectory CSVs and headless video capture.
2
3  Three video methods (choose based on your setup):
4
5      A  Xvfb framebuffer    --- Gazebo renders to a virtual display; ffmpeg
        grabs it.
6
7                                     Needs: xvfb-run, ffmpeg. Works on any server.
8
9      B  ROS camera topic    --- overhead_cam sensor publishes images;
        subscriber pipes
10
11                                     raw frames into ffmpeg. Purely headless, no
        virtual display.
12
13                                     Needs: rclpy, sensor_msgs, cv_bridge, ffmpeg.
14
15      C  Frame assembly      --- frames saved to disk during the run are
        assembled offline.
16
17                                     Needs: ffmpeg only.
18
19  """
20
21  from __future__ import annotations

```

```

13
14 import csv
15 import logging
16 import os
17 import shutil
18 import signal
19 import subprocess
20 import sys
21 import threading
22 import time
23 from pathlib import Path
24 from typing import Any, Dict, List, Optional
25
26 import yaml
27
28 logger = logging.getLogger(__name__)
29
30 _SIM_DIR    = Path(__file__).resolve().parent.parent
31 _CONFIGS_DIR = _SIM_DIR / "configs"
32 _LOGS_DIR    = _SIM_DIR / "logs"
33
34
35 # -----
36     -----
37 # Config helpers
38 # -----
39     -----
40
41 def load_config(config_path: Optional[Path] = None) -> Dict[str, Any]:
42     """Load last_run.yaml or a specified config file."""
43     path = config_path or (_CONFIGS_DIR / "last_run.yaml")
44     if not path.exists():
45         logger.error(f"Config not found: {path}")
46     return {}

```

```

45     with path.open("r") as fh:
46         return yaml.safe_load(fh) or {}
47
48
49 # -----
50 # Trajectory CSV export
51 # -----
52
53 def export_trajectory_csv(
54     robot_id: int,
55     trajectory: List[Dict[str, Any]],
56     output_dir: Path,
57 ) -> Path:
58     """Write a per-robot trajectory list to CSV.
59
60     Args:
61         robot_id: Integer robot index.
62         trajectory: List of dicts with keys: sim_time, x, y, heading,
63         task_id.
64         output_dir: Directory to write into.
65
66     Returns:
67         Path to the written CSV file.
68
69     """
70     output_dir.mkdir(parents=True, exist_ok=True)
71     out_path = output_dir / f"robot_{robot_id}_trajectory.csv"
72     fieldnames = ["sim_time", "x", "y", "heading", "task_id"]
73     with out_path.open("w", newline="") as fh:
74         writer = csv.DictWriter(fh, fieldnames=fieldnames)
75         writer.writeheader()
76         for row in trajectory:
77             writer.writerow({k: row.get(k, "") for k in fieldnames})

```

```

76     logger.info(f"Trajectory CSV: {out_path}")
77     return out_path
78
79
80 def mock_trajectories(num_robots: int, num_steps: int = 100) -> Dict[int,
    List[Dict]]:
81     """Generate mock trajectory data for testing when no bag file is
        available."""
82     import math
83     import random
84
85     trajectories: Dict[int, List[Dict]] = {}
86     for robot_id in range(num_robots):
87         traj: List[Dict] = []
88         x, y = random.uniform(-10, 10), random.uniform(-10, 10)
89         heading = random.uniform(0, 2 * math.pi)
90         for step in range(num_steps):
91             x += math.cos(heading) * 0.5
92             y += math.sin(heading) * 0.5
93             heading += random.uniform(-0.3, 0.3)
94             traj.append({
95                 "sim_time": round(step * 0.078, 4),
96                 "x":         round(x, 4),
97                 "y":         round(y, 4),
98                 "heading":   round(heading, 4),
99                 "task_id":   random.randint(0, 9),
100            })
101         trajectories[robot_id] = traj
102     return trajectories
103
104
105 # -----
    -----
106 # Method A --- Xvfb virtual framebuffer capture

```

```

107 # -----
108
109 def capture_xvfb(
110     world_path: Path,
111     output_path: Path,
112     display: str = ":99",
113     resolution: str = "1920x1080",
114     fps: int = 30,
115     duration_s: Optional[int] = None,
116 ) -> subprocess.Popen:
117     """Launch Gazebo under Xvfb and capture the virtual display with
118         ffmpeg.
119
120     This is the simplest headless video method --- no ROS camera topic
121     needed.
122
123     Requires: xvfb-run (apt install xvfb), ffmpeg, gz sim.
124
125     Usage::
126
127         proc = capture_xvfb(world_path, Path("output/sim.mp4"))
128         time.sleep(60)    # let simulation run
129         proc.terminate()
130
131     Args:
132         world_path: Path to the generated .world SDF file.
133         output_path: Destination .mp4 file.
134         display: X display number for Xvfb (default :99).
135         resolution: Virtual screen resolution (default 1920x1080).
136         fps: Output video frame rate.
137         duration_s: Stop after this many seconds (None = run until
138                     terminated).
139
140     Returns:

```

```

137         Popen handle for the ffmpeg process (Gazebo runs inside xvfb-run).
138         """
139         if not shutil.which("xvfb-run"):
140             raise RuntimeError("xvfb-run not found --- install with: sudo apt
141             install xvfb")
142         if not shutil.which("ffmpeg"):
143             raise RuntimeError("ffmpeg not found --- install with: sudo apt
144             install ffmpeg")
145
146         output_path.parent.mkdir(parents=True, exist_ok=True)
147
148         gz_cmd = [
149             "xvfb-run", "--auto-servernum",
150             "--server-args", f"-screen 0 {resolution}x24",
151             "gz", "sim", "--headless-rendering", "-r", str(world_path),
152         ]
153
154         env = os.environ.copy()
155         gz_proc = subprocess.Popen(gz_cmd, env=env)
156         logger.info(f"Gazebo launched under Xvfb (PID {gz_proc.pid})")
157
158         # Give Gazebo a moment to open the virtual display
159         time.sleep(3)
160
161         ffmpeg_cmd = [
162             "ffmpeg", "-y",
163             "-f", "x11grab",
164             "-r", str(fps),
165             "-s", resolution,
166             "-i", f"{display}.0",
167             "-c:v", "libx264",
168             "-preset", "fast",
169             "-pix_fmt", "yuv420p",
170         ]
171
172         if duration_s is not None:

```

```

169         ffmpeg_cmd += ["-t", str(duration_s)]
170         ffmpeg_cmd.append(str(output_path))
171
172         ffmpeg_proc = subprocess.Popen(ffmpeg_cmd)
173         logger.info(f"ffmpeg capturing display {display} → {output_path}")
174
175         # When ffmpeg finishes (duration reached or killed), also kill Gazebo
176         def _cleanup() -> None:
177             ffmpeg_proc.wait()
178             if gz_proc.poll() is None:
179                 gz_proc.terminate()
180
181         threading.Thread(target=_cleanup, daemon=True).start()
182         return ffmpeg_proc
183
184
185 # -----
186     -----
187 # Method B --- ROS 2 camera topic → ffmpeg pipe
188 # -----
189     -----
190
191 def capture_ros_camera_topic(
192     output_path: Path,
193     topic: str = "/overhead_cam/image_raw",
194     width: int = 1920,
195     height: int = 1080,
196     fps: int = 30,
197 ) -> Optional[subprocess.Popen]:
198     """Subscribe to the overhead camera ROS 2 topic and pipe frames into
199     ffmpeg.
200
201     The world SDF includes an overhead camera sensor at
202     ``/overhead_cam/image_raw``

```

```

199     (added by world_writer.py). The ros_gz_bridge in the launch file
200     bridges this
201
202     This function starts a helper ROS 2 node that subscribes to that
203     topic,
204     extracts raw RGB bytes, and writes them to ffmpeg via stdin pipe.
205
206     Requires: rclpy, sensor_msgs, ffmpeg.
207
208     Usage::
209
210         proc = capture_ros_camera_topic(Path("output/sim.mp4"))
211         # ... simulation runs ...
212         proc.stdin.close() # signals ffmpeg to finish and write the file
213         proc.wait()
214
215     Args:
216         output_path: Destination .mp4 file.
217         topic: ROS 2 image topic published by the overhead camera sensor.
218         width: Image width in pixels (must match SDF sensor definition).
219         height: Image height in pixels.
220         fps: Output video frame rate.
221
222     Returns:
223         Popen handle for the ffmpeg process, or None if rclpy is
224         unavailable.
225
226     """
227     try:
228         import rclpy # noqa: F401 (import check only --- actual spin in
229             subprocess)
230     except ImportError:
231         logger.error("rclpy not available --- cannot use Method B (ROS
232             camera topic)")

```



```

228         return None
229
230     if not shutil.which("ffmpeg"):
231         raise RuntimeError("ffmpeg not found --- install with: sudo apt
install ffmpeg")
232
233     output_path.parent.mkdir(parents=True, exist_ok=True)
234
235     ffmpeg_cmd = [
236         "ffmpeg", "-y",
237         "-f", "rawvideo",
238         "-vcodec", "rawvideo",
239         "-pix_fmt", "rgb24",
240         "-s", f"{width}x{height}",
241         "-r", str(fps),
242         "-i", "pipe:0",
243         "-c:v", "libx264",
244         "-preset", "fast",
245         "-pix_fmt", "yuv420p",
246         str(output_path),
247     ]
248     ffmpeg_proc = subprocess.Popen(ffmpeg_cmd, stdin=subprocess.PIPE)
249     logger.info(f"ffmpeg pipe open → {output_path}")
250
251     def _ros_subscriber_thread() -> None:
252         import rclpy
253         from rclpy.node import Node
254         from sensor_msgs.msg import Image
255
256         rclpy.init()
257
258         class FrameGrabber(Node):
259             """Minimal rclpy node that writes image frames to ffmpeg
stdin."""

```

```

260
261         def __init__(self) -> None:
262             super().__init__("frame_grabber")
263             self.create_subscription(Image, topic, self._cb, 10)
264
265         def _cb(self, msg: Image) -> None:
266             if ffmpeg_proc.stdin and ffmpeg_proc.poll() is None:
267                 try:
268                     ffmpeg_proc.stdin.write(bytes(msg.data))
269                 except BrokenPipeError:
270                     pass
271
272         node = FrameGrabber()
273         try:
274             rclpy.spin(node)
275         except Exception:
276             pass
277         finally:
278             node.destroy_node()
279             rclpy.shutdown()
280
281         t = threading.Thread(target=_ros_subscriber_thread, daemon=True)
282         t.start()
283         logger.info(f"ROS subscriber started on topic {topic}")
284         return ffmpeg_proc
285
286
287 # -----
288     -----
289 # Method C --- Assemble saved PNG frames
290 # -----
291     -----
292
293 def create_video_from_frames(

```

```

292     frames_dir: Path,
293     output_path: Path,
294     fps: int = 30,
295 ) -> bool:
296     """Assemble sequentially named PNG frames into a video with ffmpeg.
297
298     Use this when frames were saved to disk during the simulation run
299     (e.g. via 'ros2 run image_transport republish' piped to a frame
300     saver).
301
302     Frames must be named 'frame_000000.png', 'frame_000001.png', etc.
303
304     Args:
305         frames_dir: Directory containing PNG frames.
306         output_path: Destination .mp4 file.
307         fps: Output video frame rate.
308
309     Returns:
310         True on success, False on failure.
311
312     """
313     if not frames_dir.exists():
314         logger.warning(f"Frames directory not found: {frames_dir}")
315         return False
316
317     frames = sorted(frames_dir.glob("frame_*.png"))
318
319     if not frames:
320         logger.warning(f"No frame_*.png files found in {frames_dir}")
321         return False
322
323     output_path.parent.mkdir(parents=True, exist_ok=True)
324
325     cmd = [
326         "ffmpeg", "-y",
327         "-framerate", str(fps),
328         "-pattern_type", "glob",
329         "-i", str(frames_dir / "frame_*.png"),

```

```

325         "-c:v", "libx264",
326         "-pix_fmt", "yuv420p",
327         str(output_path),
328     ]
329     logger.info(f"Assembling {len(frames)} frames → {output_path}")
330     try:
331         result = subprocess.run(cmd, capture_output=True, text=True,
332                                 timeout=600)
333         if result.returncode == 0:
334             logger.info(f"Video written: {output_path}")
335             return True
336         logger.error(f"ffmpeg failed:\n{result.stderr}")
337         return False
338     except FileNotFoundError:
339         logger.error("ffmpeg not found --- sudo apt install ffmpeg")
340         return False
341     except subprocess.TimeoutExpired:
342         logger.error("ffmpeg timed out after 600 s")
343         return False
344
345 # -----
346 # -----
347 # Frame-saver helper (used with Method C)
348 # -----
349
350 def save_frames_from_topic(
351     frames_dir: Path,
352     topic: str = "/overhead_cam/image_raw",
353     max_frames: int = 10_000,
354 ) -> None:
355     """Subscribe to the camera topic and save each frame as a PNG file.

```

```

356     Runs synchronously until interrupted or max_frames reached.
357     Intended to be called in a subprocess or thread before
        create_video_from_frames().
358
359     Requires: rclpy, sensor_msgs, cv_bridge, opencv-python.
360     """
361     try:
362         import rclpy
363         from rclpy.node import Node
364         from sensor_msgs.msg import Image
365         import cv2
366         from cv_bridge import CvBridge
367     except ImportError as exc:
368         logger.error(f"Missing dependency for frame saver: {exc}")
369         logger.error("Install: pip install opencv-python && sudo apt
        install ros-jazzy-cv-bridge")
370         return
371
372     frames_dir.mkdir(parents=True, exist_ok=True)
373     bridge = CvBridge()
374     frame_count = 0
375
376     rclpy.init()
377
378     class FrameSaver(Node):
379         """rclpy node that writes camera images to PNG files."""
380
381         def __init__(self) -> None:
382             super().__init__("frame_saver")
383             self.create_subscription(Image, topic, self._cb, 10)
384
385         def _cb(self, msg: Image) -> None:
386             nonlocal frame_count
387             if frame_count >= max_frames:

```

```

388         return
389         cv_img = bridge.imgmsg_to_cv2(msg, desired_encoding="bgr8")
390         fname = frames_dir / f"frame_{frame_count:06d}.png"
391         cv2.imwrite(str(fname), cv_img)
392         frame_count += 1
393         if frame_count % 100 == 0:
394             logger.info(f"Saved {frame_count} frames to {frames_dir}")
395
396     node = FrameSaver()
397     try:
398         rclpy.spin(node)
399     except KeyboardInterrupt:
400         pass
401     finally:
402         node.destroy_node()
403         rclpy.shutdown()
404     logger.info(f"Frame saver stopped. Total frames: {frame_count}")
405
406
407 # -----
408     -----
409 # CLI entry point
410 # -----
411     -----
412
413 def main() -> None:
414     """CLI: python export_recording.py [config_path] [--method A|B|C].
415
416     Reads last_run.yaml (or the path you pass) and exports based on the
417     capture_video flag and chosen method.
418
419     Examples::

```

```

419         # Export CSVs + assemble frames already on disk (Method C,
default):
420         python Simulation/scripts/export_recording.py
421
422         # Capture from ROS camera topic while sim is running (Method B):
423         python Simulation/scripts/export_recording.py
configs/last_run.yaml --method B
424
425     """
426     import argparse
427
428     logging.basicConfig(level=logging.INFO, format="%(levelname)s
%(message)s")
429
430     parser = argparse.ArgumentParser(description="AQUA Guardian recording
exporter")
431     parser.add_argument("config", nargs="?", default=None, help="Path to
config YAML")
432     parser.add_argument(
433         "--method", choices=["A", "B", "C"], default="C",
434         help="Video capture method: A=Xvfb, B=ROS topic pipe, C=frame
assembly"
435     )
436     args = parser.parse_args()
437
438     config_path = Path(args.config) if args.config else None
439     config = load_config(config_path)
440     if not config:
441         logger.error("Empty or missing config --- aborting")
442         sys.exit(1)
443
444     num_robots      = config.get("num_robots", 4)
445     scenario        = config.get("scenario", "warehouse")

```

```

446     output_folder = Path(config.get("output_folder", str(_LOGS_DIR /
447         scenario)))
448     save_csv       = config.get("save_trajectory_csv", True)
449     capture_video  = config.get("capture_video", False)
450
451     logger.info(f"Scenario: {scenario}, robots: {num_robots}, method:
452         {args.method}")
453
454     # -- Trajectory CSVs -----
455     -----
456     if save_csv:
457         try:
458             from rosbag2_py import SequentialReader # type:
459             ignore[import]
460             logger.info("rosbag2_py available --- reading from bag (not
461             yet implemented)")
462             trajectories = mock_trajectories(num_robots)
463         except ImportError:
464             logger.warning("rosbag2_py not available --- using mock
465             trajectory data")
466             trajectories = mock_trajectories(num_robots)
467
468             csv_dir = output_folder / "trajectories"
469             for robot_id, traj in trajectories.items():
470                 export_trajectory_csv(robot_id, traj, csv_dir)
471
472     # -- Video capture -----
473     -----
474     if capture_video:
475         video_path = output_folder / f"{scenario}_sim.mp4"
476
477         if args.method == "A":
478             # Xufb: user must provide world_path separately
479             world_file = config.get("last_world_file")

```



```

473         if not world_file:
474             logger.error(
475                 "Method A requires 'last_world_file' in config. "
476                 "It is saved automatically when you click Simulate."
477             )
478             sys.exit(1)
479             logger.info("Starting Xvfb + ffmpeg capture --- press Ctrl-C
to stop")
480             proc = capture_xvfb(Path(world_file), video_path)
481             try:
482                 proc.wait()
483             except KeyboardInterrupt:
484                 proc.send_signal(signal.SIGINT)
485
486         elif args.method == "B":
487             logger.info("Subscribing to /overhead_cam/image_raw --- press
Ctrl-C to stop")
488             proc = capture_ros_camera_topic(video_path)
489             if proc is None:
490                 sys.exit(1)
491             try:
492                 while proc.poll() is None:
493                     time.sleep(1)
494             except KeyboardInterrupt:
495                 if proc.stdin:
496                     proc.stdin.close()
497                 proc.wait()
498
499         else: # Method C
500             frames_dir = output_folder / "frames"
501             success = create_video_from_frames(frames_dir, video_path)
502             if not success:
503                 logger.warning("Video assembly skipped or failed ---
check frames_dir")

```

```
504
505     logger.info("Export complete.")
506
507
508 if __name__ == "__main__":
509     main()
```

Listing C.35: Post-simulation export utility. Writes per-robot trajectory CSVs from recorded state data. Supports three headless video capture methods: A — Xvfb virtual framebuffer (`xvfb-run + ffmpeg`); B — ROS 2 overhead camera topic piped to `ffmpeg`; C — offline assembly of PNG frames saved during the run. Method is selected via `--method A|B|C` on the command line.