UNIT AND INTEGRATION TESTING COPILOT

Group Members

Faizan Ali (01-131222-017)

Salah-ud-din Sani (01-131222-043)

Supervisor:

Dr. Tamim Ahmed Khan

A Final Year Project submitted to the Department of Software Engineering, Faculty of Engineering Sciences, Bahria University, Islamabad in the partial fulfillment for the award of degree in Bachelor of Software Engineering

May 2026

FYP Completion Certificate

Student Name: Faizan Ali **Enrolment No:** 01-131222-017

Student Name: Salah-ud-din Sani **Enrolment No:** 01-131222-043

Program of Study: Bachelor of Software Engineering

Project Title: Unit and Integration Testing Copilot

It is to certify that the above students' project has been completed to my satisfaction and to my belief, its standard is appropriate for submission for evaluation. I have also conducted a plagiarism test of this thesis using HEC prescribed software and found a similarity index at 6% that is within the permissible limit set by the HEC. I have also found the thesis in a format recognized by the department.

Supervisor's Signature: _____

Date: May 16, 2026 **Name:** Dr. Tamim Ahmed Khan

Certificate of Originality

This is to certify that the intellectual contents of the project **"Unit and Integration Testing Copilot"** are the product of my/our own work except, as cited properly and accurately in the acknowledgements and references, the material taken from such sources as research journals, books, internet, etc., solely to support, elaborate, compare, extend and/or implement the earlier work.

Further, this work has not been submitted by me/us previously for any degree, nor shall it be submitted by me/us in the future for obtaining any degree from this University, or any other university or institution. The incorrectness of this information, if proved at any stage, shall authorize the University to cancel my/our degree.

Name of the Student: Faizan Ali

Signature: _____ **Date:** May 16, 2026

Name of the Student: Salah-ud-din Sani

Signature: _____ **Date:** May 16, 2026

Abstract

Software development consistently requires testing at every stage; however, writing unit and integration tests manually is often repetitive, time-consuming, and difficult to maintain consistently across multiple files. As codebases grow, developers may unintentionally overlook certain parts of testing, especially under tight deadlines. This creates a need for practical tools that can simplify the process of building, organizing, and executing tests more efficiently without adding unnecessary complexity.

This project introduces **Unit and Integration Testing Copilot**, a locally running desktop-based assistant designed specifically for test-related tasks. Unlike cloud-based solutions, the system operates directly within the developer's environment, enabling seamless interaction with source code. It allows users to import files with a single action, after which the system performs structured analysis by parsing the code layout in detail. Based on observed patterns, it generates initial test structures that evolve according to the relationships within the code, helping developers construct validations for both small functions and larger modules.

The system further enhances the testing workflow by reducing repetitive typing, minimizing errors, and providing timely suggestions as development progresses. It adapts to changes in code structure and supports the creation of consistent and meaningful tests. By illustrating how different components of the code are connected, it simplifies the process of understanding and verifying system behavior. This makes it particularly useful for both individual learners and development teams, as it reduces the effort spent on deciding what to test while promoting a more structured and efficient testing process.

The final outcome of this project is a smart testing support tool that contributes to better software quality, increased efficiency, and improved confidence in the reliability of software systems.

Keywords: Software Testing, Unit Testing, Integration Testing, Test Automation, Intelligent Assistant, Desktop Application

Dedication

First, we dedicate this project to Almighty Allah, the Most Merciful and the Most Beneficent, who gave us the strength, knowledge, and ability to complete this work successfully.

We dedicate this work to our beloved parents for their unconditional love, continuous support, prayers, and encouragement throughout our academic journey.

We also dedicate this project to our respected supervisor, **Dr. Tamim Ahmed Khan**, whose guidance and support played a vital role in the successful completion of this project.

Acknowledgement

We would like to express our sincere gratitude to our respected supervisor, **Dr. Tamim Ahmed Khan**, for his valuable guidance, continuous support, and insightful feedback throughout the development of this project. His supervision played a crucial role in shaping and completing this work successfully.

We are also thankful to the faculty members of the Department of Software Engineering, Bahria University, for providing us with the knowledge and academic foundation necessary for this project.

We acknowledge the support and encouragement provided by our colleagues and friends, which contributed positively during the course of this work.

PROJECT TITLE

Unit and Integration Testing Copilot

Sustainable Development Goals

(Please tick the relevant SDG(s) linked with FYP)

SDG No	Description of SDG	SDG No	Description of SDG
☐ SDG 1	No Poverty	■ SDG 9	Industry, Innovation, and Infrastructure
☐ SDG 2	Zero Hunger	☐ SDG 10	Reduced Inequalities
☐ SDG 3	Good Health and Well Being	☐ SDG 11	Sustainable Cities and Communities
☐ SDG 4	Quality Education	☐ SDG 12	Responsible Consumption and Production
☐ SDG 5	Gender Equality	☐ SDG 13	Climate Change
☐ SDG 6	Clean Water and Sanitation	☐ SDG 14	Life Below Water
☐ SDG 7	Affordable and Clean Energy	☐ SDG 15	Life on Land
☐ SDG 8	Decent Work and Economic Growth	☐ SDG 16	Peace, Justice and Strong Institutions
		☐ SDG 17	Partnerships for the Goals

Range of Complex Problem Solving

Sr. No.	Attribute	Complex Problem	Tick
1	Range of conflicting requirements	Involve wide-ranging or conflicting technical, engineering and other issues.	✓
2	Depth of analysis required	Have no obvious solution and require abstract thinking, originality in analysis to formulate suitable models.	✓
3	Depth of knowledge required	Requires research-based knowledge much of which is at, or informed by, the forefront of the professional discipline and which allows a fundamentals-based, first principles analytical approach.	✓
4	Familiarity of issues	Involve infrequently encountered issues.	✓
5	Extent of applicable codes	Are outside problems encompassed by standards and codes of practice for professional engineering.	☐
6	Extent of stakeholder involvement and level of conflicting requirements	Involve diverse groups of stakeholders with widely varying needs.	✓
7	Consequences	Have significant consequences in a range of contexts.	☐
8	Interdependence	Are high level problems including many component parts or sub-problems.	✓

Range of Complex Problem Activities

Sr. No.	Attribute	Complex Activities	Tick
1	Range of resources	Involve the use of diverse resources (and for this purpose, resources include people, money, equipment, materials, information and technologies).	☐
2	Level of interaction	Require resolution of significant problems arising from interactions between wide ranging and conflicting technical, engineering or other issues.	☐
3	Innovation	Involve creative use of engineering principles and research-based knowledge in novel ways.	✓
4	Consequences to society and the environment	Have significant consequences in a range of contexts, characterized by difficulty of prediction and mitigation.	☐
5	Familiarity	Can extend beyond previous experiences by applying principles-based approaches.	✓

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Chapter 1

Introduction

1.1 Purpose

Unit and Integration Testing Copilot is an intelligent copilot tool which should be used as a desktop tool allowing programmers to analyze source code and create tests. According to common software engineering practices, testing usually takes much time since it is required to conduct manual analysis and test writing.

To solve such challenges, it is suggested to implement the following tool based on automated code analysis capabilities and intelligent algorithms used for creation of unit and integration tests. With the help of these algorithms, the code will be analyzed automatically; possible mistakes will be detected, and adequate tests will be created.

The suggested software solution should contain the following three modules:

1.1.1 User Interface (Frontend)

The software has an easy-to-use desktop interface where programmers have the option of uploading or writing their source codes. Furthermore, one will be able to analyze the results of the tests and also see how well their code and tests performed.

1.1.2 Core Processing Engine (Backend)

This module manages all the actions that are required to execute various functions, such as coding analysis and parsing. It also manages the interaction between processes for effective execution of the system.

1.1.3 Analysis and Intelligence Layer

For this module, the focus will be on the analysis of code patterns to see what sections need testing. Through this, we shall generate our test cases and get to know more about the code itself.

The main purpose of the Unit and Integration Testing Copilot is to make testing as efficient as possible by automation and efficient generation of tests as well as developing a robust system. The main objective of this paper is to provide information for the stakeholders in the system.

1.2 Document Conventions

Below is an outline of the standards that have been set out for use within this SRS document. These standards will help all those involved interpret the document in the same way.

1.2.1 Formatting Standards

The important new words used have been highlighted in bold face. This will make the text easy to read and understand as it will show the most important points in it.

System components and modules are written using consistent naming to clearly identify different parts of the system.

Code-related elements such as file names, variables, and programming constructs are represented in a distinct format to differentiate them from normal text.

User interactions and actions are described clearly to indicate how users will interact with the system.

1.2.2 Terminology and Acronyms

The following terms and acronyms are used throughout this document:

- **SRS (Software Requirements Specification):** A document that defines the functional and non-functional requirements of the system.
- **FYP (Final Year Project):** Refers to the academic project undertaken as part of the degree program.
- **LLM (Large Language Model):** A model used to assist in generating test cases and providing intelligent suggestions.
- **API (Application Programming Interface):** The interface used for communication between different components of the system.
- **Copilot:** Refers to the test generation functionality of the system.

1.2.3 Requirement Identification

Requirements in this document are categorized to clearly distinguish their purpose:

- **Functional Requirements (FR):** Describe the system's behavior, including inputs, processing, and outputs.
- **Non-Functional Requirements (NFR):** Describe quality attributes such as performance, usability, and reliability.
- **Data and Deployment Requirements (DDR):** Describe data handling, storage, and deployment aspects of the system.

Each requirement is assigned a unique identifier to make it easier to reference and manage throughout the development process.

1.2.4 Requirement Priority Levels

Requirements are prioritized based on their importance to the system:

- **Critical:** Essential for system functionality. Without these, the system cannot operate.
- **High:** Important features that significantly improve the system and are expected to be implemented.

- **Medium:** Features that enhance usability or efficiency but are not mandatory.
- **Low:** Features that are not required in the current phase but may be considered in future development.

1.3 Product Scope

With this system, one can create a setup for analyzing source codes, testing, and retrieving information regarding testing. The application has been developed to minimize the workload required for generating the tests. With the help of this system, the developers will be able to learn about certain aspects of the application development process and how the code behaves.

The scope of this project is confined to helping developers generate tests and analyze the codes within the desktop application. This system is not meant to compete with the developer's ability but will only work as a tool that enhances productivity while developing the code.

The scope of this project is limited to assisting in test generation and basic code analysis within a desktop application environment. It does not replace the developer but acts as a support tool to improve productivity and code quality.

1.3.1 Product Description

The Copilot for Unit and Integration Testing acts as a supporting software application to assist developers by performing source code analyses and construction of tests. The software application provides users with an interface through which codes can be imported or written, analyzed, and where the output generated from the unit and integration testing will be obtained.

The software application will focus on identifying the source codes' regions which need tests and support in developing appropriate test cases.

1.3.2 Goals and Objectives

Some of the objectives of this project are:

- To make the process of creating unit and integration test cases very simple.

- To help software programmers detect faults in their source code early on.
- To improve software quality through the practice of effective testing.
- To save time spent writing and maintaining test cases.
- To create a structured platform for testing.

1.3.3 Scope Definition

The scope of the proposed project will be focused on development of the software capable of performing code analysis and generating test cases. In particular, the program will have an interface that will allow developers to enter their code or upload it from an external source file, analyze the entered code for errors and generate test cases.

Backend functionality will be oriented towards processing of the obtained data and control of the test case generation process. Additionally, the output of the system will be structured, allowing users to understand the results of the test and the behavior of the code under test.

In summary, the key aim of the project is to unite all the above-mentioned features in a single platform that will simplify the testing process.

However, it should be noted that the current version of the project is only focused on development of basic testing tools. The project will allow conducting unit and integration testing but will not be able to perform large-scale automation, performance, security and load testing tasks. These issues will be addressed in further versions of the product.

CHAPTER 2

LITERATURE REVIEW

Chapter 2

Literature Review

One may say that literature review is an inseparable component of any research work since through it, one can evaluate what has already been done in this field. The primary aim of conducting literature reviews is identifying existing approaches to problem-solving, their deficiencies, and providing recommendations for overcoming them. There are many tools used to help developers create test cases. However, the use of such tools requires much effort on the part of users.

2.1 Unit Testing Tools

In [1], the authors have taken into account the different strategies that are used for the validation of the components using testing tools like JUnit and NUnit. Through such testing tools, the developer will be able to write tests for testing components or functions to ensure that they operate properly. Even though JUnit and NUnit are very reliable testing tools, the use of these tools will require you to create your test cases.

2.2 Automated Test Generation

There [2] have been investigations into the topic of test case generation, which aims to cut down the time involved in developing test cases. The method used in this context involves the examination of the source code to create tests based on the nature of the program. While it may simplify the

testing process, the method is quite inflexible and fails to handle component interactions effectively.

2.3 Integration Testing Approaches

The methodology in [3] focused on integration testing, and here the key point was that interaction testing was required between various modules of a software product. The approach adopted included testing the processes of data communication among different components. However, integration testing proves to be quite challenging because of significant configuration required.

2.4 Mutation Testing

According to [4], mutation testing has been used for the evaluation of the effectiveness of the test cases. The process involves making minor changes in the software and checking if the existing test cases are able to identify those changes. Mutation testing is a highly effective method of detecting defects in the test cases of the software. Nevertheless, due to its high computation cost, it cannot be performed under normal conditions.

2.5 Developer Tools and IDE Support

In [5], it can be seen that there have been several research studies highlighting the effectiveness of IDEs like Visual Studio Code, which offers multiple features including code navigation, debugging, and automation testing. However, the feature of test execution in IDEs is provided by external extensions rather than being an integrated part of the package.

2.6 Limitations of Existing Systems

While taking into account the existing technologies and tools, it becomes clear that most of the solutions focus on just part of the testing procedures, namely, unit testing and integration testing. Most of the tools developed today do not have a fully integrated testing system that covers all the aspects

of testing within one tool. Furthermore, many tools still require a lot of effort and cannot generate and analyze test cases automatically.

2.7 Motivation for Proposed System

Taking into account the limitations inherent to existing systems, it can be said that there is obviously a need for a platform that combines code analysis, generation, and testing capabilities all together. It comes in the form of Unit and Integration Testing Copilot, a desktop application that supports the process of uploading, analysis, and generating of different test cases related to the given piece of code.

CHAPTER 3

Requirement Specification

Chapter 3

Requirement Specification

3.1 Product Perspective

UITC stands for Unit and Integration Testing Copilot – a desktop application designed to enhance the effectiveness of traditional techniques used in unit and integration testing. As described earlier, conventional methods involve manual codebase analysis and test creation through the use of auxiliary software. Hence, the process takes some time and varies significantly depending on the developer's competence.

UITC helps overcome the existing drawbacks thanks to the unified platform developed using Angular and Electron technologies. Thus, the solution involves a desktop application featuring an intuitive user interface, much like modern code editors. Furthermore, the application allows uploading the codebase, getting test cases and their analytics without relying on any additional software.

According to the requirements, the copilot should provide assistance in performing unit tests, integration tests, and mutation testing. Therefore, the proposed solution offers a detailed analysis of the code and Abstract Syntax Tree (AST) generation. This step helps identify dependencies in the program and create relevant and logical test cases.

Moreover, the copilot uses Cerebras AI algorithms to optimize the testing process and interact with the user. The latter is achieved through a chatbot which responds to the user's queries related to testing.

3.1.1 Key Features of the System

- **Automated Test Case Generation:** This tool helps you create unit, integration, and mutation tests automatically based on the code uploaded by you, hence reducing your time and allowing you to increase the scope of your test coverage.
- **AST-Based Integration Testing:** Integration tests are created using AST analysis, which makes the platform recognize how different modules, functions, and components interact.
- **Mutation Testing Support:**

  Mutations are performed through this platform using your code, making the platform understand how effectively the tests created can detect any mutations.
- **Test Coverage and Reporting:** You receive full coverage data and detailed reports, which help you assess the quality and test coverage of your test.

  .
- **Interactive Chatbot Interface:** With the help of Cerebras AI chatbot, you can interact with the platform and request generation of tests or analyze the results.
- **IDE-like User Interface:** The desktop application comes with a UI similar to Visual Studio Code.f testing

3.1.2 System Integration Perspective

The UITC will be developed as a standalone desktop application and will be seamlessly incorporated into the software developer's process of work. The reason is that with the UITC, which offers a variety of functions, including code analysis, test generation, and reporting, it will not be required for software developers to switch between several programs for these actions.

The application offers a streamlined test process, wherein users are able to upload the code, perform the analyses, generate tests, and even evaluate the results using just one application. Not only does this solution boost

productivity but it also makes testing simpler, helping maintain consistency while testing.

By automating various processes related to software testing and providing a single point of contact, the UNITC can help improve software quality considerably.

3.2 Product Functions

The Unit and Integration Testing Copilot (UITC) provides a set of automated functions that assist developers in generating and evaluating software tests. The system focuses on simplifying the testing workflow by integrating code analysis, test generation, and result evaluation into a single platform.

The major functional components of the system are described below.

3.2.1 User Authentication

The system includes a basic authentication mechanism to control access to the application.

- **Secure Login:** Users log in using a username and password to ensure that only authorized individuals can access the system.

3.2.2 Code Analysis

The system analyzes the uploaded codebase to understand its internal structure and dependencies.

- **Code Parsing:** The system processes the source code and identifies key elements such as functions, classes, and modules.
- **AST-Based Analysis:** Abstract Syntax Tree (AST) techniques are used to determine relationships between different components of the code, which supports further testing operations.

3.2.3 Test Case Generation

The system automates the creation of different types of test cases based on the analyzed code.

- **Unit Test Generation:** Test cases are generated for individual functions and components to validate their correctness.
- **Integration Test Generation:** The system generates integration test cases by analyzing interactions between modules using AST-based relationships.
- **Testing for mutation:** Small changes will be introduced in your code to find out if the test cases generated can catch the changes. In this way, you'll know about the effectiveness of the tests that you've made.
- **Options for exporting:** All the test cases generated can be exported using the methods of copying, reviewing, and downloading.

3.2.4 Test Coverage and Reporting

This involves analyzing the code by using the coverage metrics generated by the software in order to determine the percentage of coverage of your tests on the code.

- **Coverage Test Analysis:** This involves analyzing the code by using the coverage metrics generated by the software in order to determine the percentage of coverage of your tests on the code.
- **Report Compilation:** Results of your test are organized and presented in the form of a report.

3.2.5 Chatbot Interaction

A chatbot interface is also provided to increase the ease-of-use of the software.
.

- **Integration of Cerebras AI:** The user can interact with the chatbot interface to request that the tests be generated or even get your test scores
- **Command Interface:** Actions such as test generation and retrieving test scores can be performed via simple commands issued to the system.

3.3 User Classes and Characteristics

The Unit and Integration Testing Copilot (UITC) is built to support different types of users with various technical skills and responsibilities. By identifying these specific user classes, we make sure the system provides an efficient, organized testing environment that fits the needs of software development teams.

3.3.1 Developer Users

Characteristics:

- Individuals responsible for writing, modifying, and maintaining source code.
- Possess strong programming knowledge and familiarity with development tools and testing concepts.
- Focus on implementing features, fixing defects, and maintaining code quality.

Primary Goals:

- Automatically generate unit and integration test cases to reduce manual effort.
- Evaluate test coverage to ensure critical parts of the code are tested.
- Understand dependencies and interactions between components through system insights.
- Improve code quality based on test results and analysis.

3.3.2 Tester Users

Characteristics:

- Responsible for validating software quality and reviewing test results.
- Have knowledge of testing principles such as coverage and validation techniques.

- Focus on ensuring reliability rather than modifying core application code.

Primary Goals:

- Review and validate generated test cases for accuracy and completeness.
- Assess testing effectiveness using coverage metrics and mutation testing results.
- Ensure that system functionality is thoroughly tested.
- Provide feedback to improve testing quality.

3.4 Operating Environment

3.4.1 Hardware Platform

- **Client Systems:** Desktop or laptop systems with a minimum of 8GB RAM and a modern multi-core processor.
- Systems with higher specifications provide better performance when handling large codebases.

3.4.2 Operating System

- Supported platforms include Windows, macOS, and Linux (Ubuntu or equivalent distributions).

3.4.3 Software Components

- **Desktop Application Framework:** Angular for frontend and Electron for desktop integration.
- **Code Analysis Engine:** Abstract Syntax Tree (AST) based processing.
- **Testing Modules:** Supports unit testing, integration testing, and mutation testing.

- **AI Integration:** Cerebras AI for chatbot interaction and test case generation.
- **Programming Language Support:** The system supports test generation and execution for five programming languages.

3.4.4 Network Requirements

- A stable internet connection is required for interaction with Cerebras AI.
- Recommended minimum internet speed of 10 Mbps for smooth test generation and communication.

3.4.5 Compatibility

- Supports multiple programming languages for test generation and execution.
- Operates as a standalone desktop application without dependency on web browsers.

3.5 Design and Implementation Constraints

3.5.1 Technological Constraints

- Limited to technologies compatible with Angular and Electron frameworks.
- AST-based analysis must be supported for selected programming languages.

3.5.2 Hardware Limitations

- Performance depends on system specifications, especially for large codebases.
- Lower-end systems may experience slower processing and execution times.

3.5.3 Security

- Implements secure authentication mechanisms to restrict unauthorized access.
- Ensures safe handling of user data during processing.

3.5.4 Regulatory Policies

- Follows standard data handling practices to maintain user privacy.

3.5.5 Standards for Development

- Code development follows standard coding practices for readability and maintainability.
- Documentation is structured according to software requirement specification (SRS) guidelines.

3.6 Assumptions and Dependencies

Assumptions

- Users have basic computer literacy and are familiar with development environments and code editors.
- Users will download the application from the official website, install it on their local system, and use it as a desktop application.
- The system is installed on a compatible desktop environment with sufficient hardware resources to support code analysis and test execution.
- A stable internet connection is available, as the system relies on Cerebras AI for generating test cases and chatbot interaction.
- Users provide valid and well-structured source code to ensure accurate analysis and meaningful test generation.

Dependencies

- Cerebras AI service for generating test cases and enabling chatbot-based interaction.
- Angular and Electron frameworks for building and running the desktop application.
- Abstract Syntax Tree (AST) processing libraries for analyzing code structure and identifying relationships between components.
- Language-specific runtime environments and testing frameworks required for generating and executing test cases for the supported programming languages (currently five languages).

3.7 External Interface Requirements

3.7.1 User Interfaces

The system provides a desktop-based user interface designed to resemble a modern code editor, allowing efficient interaction for developers and testers.

Login Interface

- A simple interface for entering username and password.
- Provides secure access to the application after installation.

Main Workspace (IDE-like Interface)

- Central working area where users can view and interact with the uploaded codebase.
- Includes file navigation and code viewing panels.
- Displays generated test cases and testing results within the same interface.

Code Upload and Selection

- Allows users to upload or import a codebase into the system.
- Provides options to select specific files or modules for testing.

Test Case Management

- Displays automatically generated unit, integration, and mutation test cases.
- Allows users to view, copy, or export test cases for further use.

Test Results and Reports

- Shows results of executed test cases.
- Displays test coverage information and summary reports.

Chatbot Interface

- Provides a conversational interface powered by Cerebras AI.
- Allows users to request test generation and access results through simple commands.

3.7.2 Hardware Interfaces

- Desktop or laptop systems running Windows, macOS, or Linux.
- Minimum 8GB RAM recommended for efficient handling of large codebases.
- Standard input/output devices such as keyboard, mouse, and display.

3.7.3 Software Interfaces

- Angular framework for building the user interface.
- Electron framework for desktop application functionality.
- AST-based libraries for code analysis.

- Cerebras AI for test generation and chatbot interaction.
- Language-specific testing frameworks for executing generated test cases.

Data Flow:

- The user downloads and installs the application from the website.
- The user uploads code into the system.
- The system analyzes the code using AST-based techniques.
- Requests are sent to Cerebras AI for test case generation.
- Generated test cases are returned and displayed in the interface.
- Test cases are executed, and results are presented in reports.

3.7.4 Communication Interfaces

- Internet-based communication for interacting with Cerebras AI services.
- Secure protocols (such as HTTPS) for transmitting data to external services.
- Internal communication between application modules managed within the Electron environment.

The use case diagrams help in showing the connection that exists between users and the system known as UITC. This diagram shows all the features in the application including coding and generating tests.

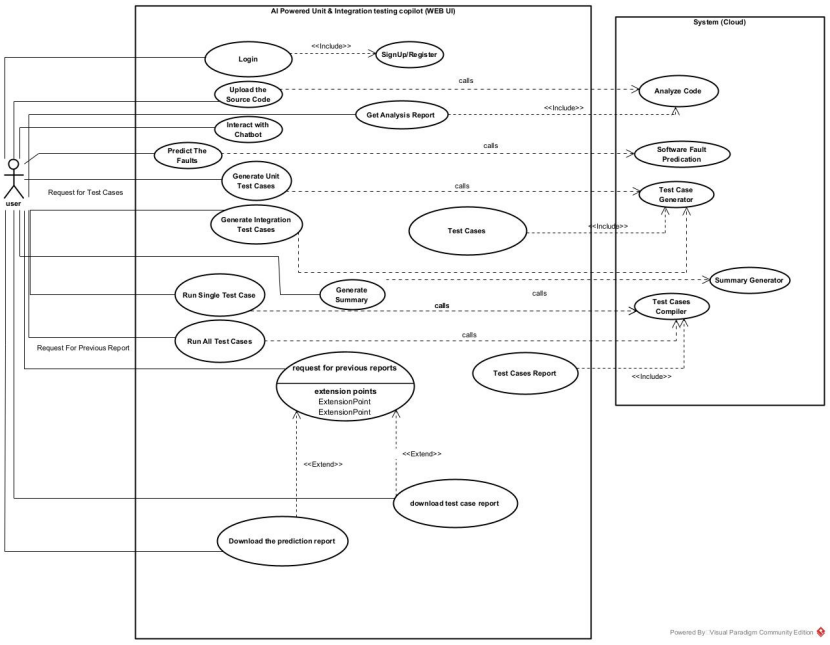


Figure 3.1: Use Case Diagram

3.7.5 BPMN Diagrams

3.7.6 Use Case Diagram

BPMN is employed to show the process flow in the process of unit and integration testing copilot (UITC). In the diagrams created using BPMN, the process that a user goes through when using the software in file handling, AI generated testing or coding, among others, is described accurately.

File Upload and Management Process

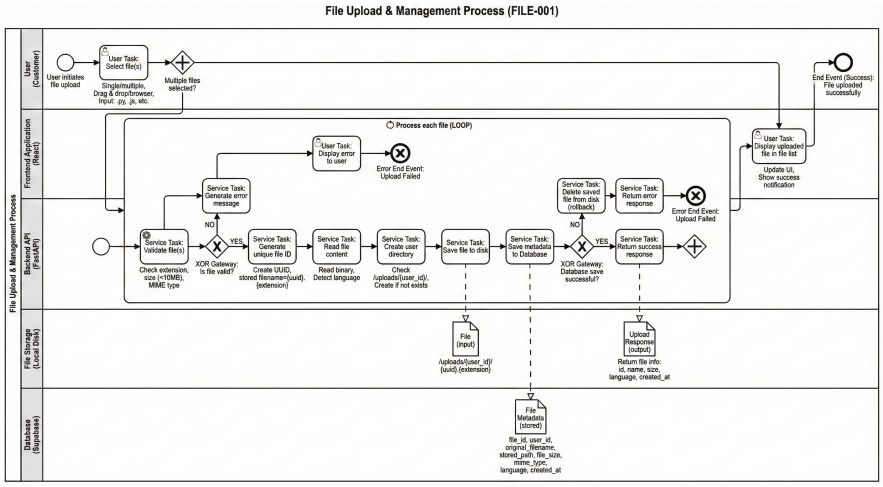


Figure 3.2: BPMN Diagram for File Upload and Management Process

AI Test Generation Process

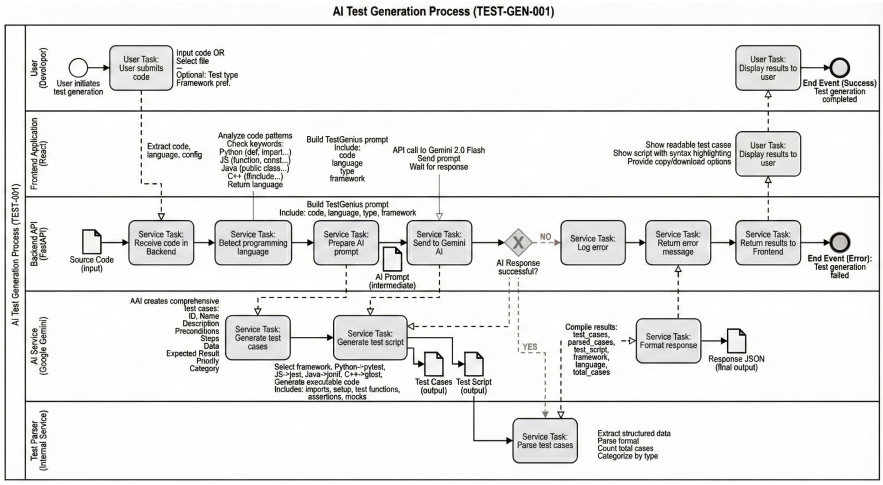


Figure 3.3: BPMN Diagram for AI Test Generation Process

Code and Test Execution Process

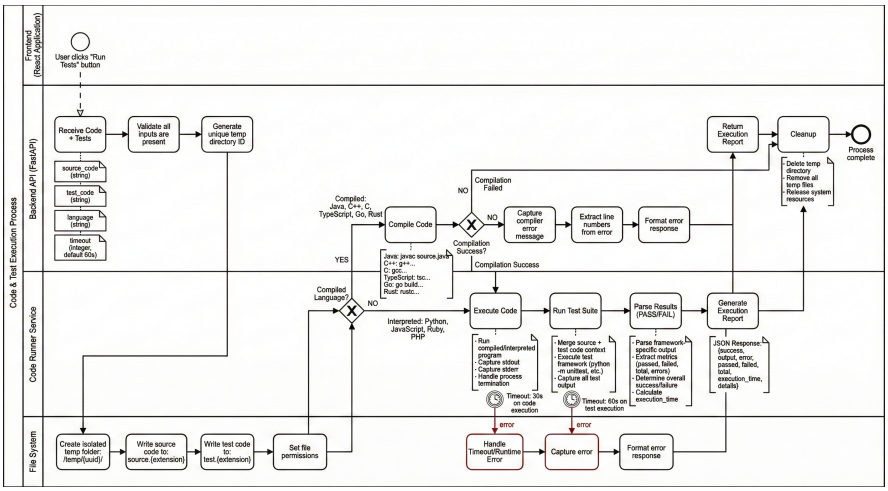


Figure 3.4: BPMN Diagram for Code and Test Execution Process

3.8 Non-Functional Requirements

3.8.1 Performance

Response Time and Efficiency

Another crucial characteristic that speaks of the efficiency of Unit and Integration Testing Copilot is fast code processing and obtaining results. The program is made to analyze, generate, and run tests at minimum speed loss so that you could work productively without any pauses. If you have huge code, the software will be able to process it fast as well.

User Feedback Mechanism

Moreover, giving feedback is important for the efficiency of application usage. If the program finds any mistakes in your code or notices anything preventing the test from being executed correctly, it reports that. As opposed to most programs, the one under discussion informs about any issues faced clearly.

Processing Optimization

For efficient functioning, the app uses structured code analysis based on AST processing. It guarantees speedy code analysis and efficient generation of tests. Moreover, it is resource-saving so that you would not need too much time and computing capacities to work effectively.

3.8.2 Scalability

Handling of Code Complexity

The software will be able to process code with varying length – from small programs to complex multi-file programming projects. The software ensures that the efficiency is not affected by increased complexity or length of the program code.

Multi-Language Support

Currently, the system can support a number of different programming languages, meaning that the testing will be conducted on programs written in five programming languages.

Extensibility

It should be noted that this system is ready for further development, allowing to easily expand it by adding support for additional programming languages and methods of analysis.

3.8.3 Reliability

System Stability

Stability is a critical characteristic of any development tool. This ensures that the software runs efficiently and does not crash randomly in all cases, including during code analysis and testing.

Robust Error Handling

The software takes into account the likelihood of unexpected user behavior, which may include incorrectly typed code, syntactic errors, or unrecognized

patterns. However, even when these happen, the software will handle them sufficiently well to ensure that they do not affect your work.

Recovery Capability

If the software fails, it will be able to reset itself and resume processing without losing any data gathered by your input until then...

3.8.4 Usability

Interface Design

The product is delivered with the user interface based on the principles of modern code editors, thus ensuring usability and good organization of the process. Indeed, the interface of the kind reduces the period necessary to adapt to its usage through the natural interaction at once.

Intelligent Assistance

Moreover, the product comes with the embedded chatbot powered by Cerebras AI technology. The functions provided by this chatbot facilitate working with the application due to the availability of natural language commands for controlling it.

Accessibility of the Application

The system is distributed through an official platform from which users can download and install it locally. Once installed, all features are accessible within a self-contained desktop environment.

Output and Reporting

Finally, the product comes from a reliable source and is available for downloading and installation locally. After the installation is completed, users will get access to all capabilities of the application within their desktop.

In particular, it provides the following outputs:

- Test cases generated by the application
- Execution results of the tests

- Coverage reports

3.8.5 Maintainability

Modular Architecture

The design uses different independent components such as UI, code analysis engine, test generator, and execution module. The ease of modifying any component independently allows for an easy upgrade of the entire system in the future.

Logging and Monitoring

The system maintains a record of all actions performed and their results. This record may include logs of errors encountered during operations of the system. This makes debugging the system quite easy.

System Updates

The system has been designed to allow easy upgrades even in the future.

3.8.6 Interoperability

Data Handling

The program caters to data file types commonly used by allowing the user to export test cases and report results, which then makes it easier for them to integrate their outputs in other applications or documentation.

Integration Potential

Though at the moment it functions as a stand-alone program, the tool was designed in a way that it will be possible to integrate the product into other third-party applications in the future.

Platform Compatibility

The program works just as effectively on different operating systems such as Windows, Macintosh, and Linux..

3.8.7 Availability

System Access

Once installed, the software will always be accessible locally, which means that the application can be used independently without the need to depend on another platform.

Dependency on Network Services

Some of the features, such as creating tests and engaging in chatbots conversations via Cerebras AI, require an internet connection to perform their tasks.

Continuous Operation

The application enables the performance of constant functions like conducting analyses, creating tests, and reviewing results. There are no disruptions except when it is not possible to access certain internet-based features.

3.8.8 Technical Feasibility

Feasibility Overview

From the technical perspective, the usage of the Unit and Integration Testing Copilot tool is technically feasible due to the presence of well-established and widely adopted technologies.

Technology Stack

Firstly, the utilization of such libraries as Angular and Electron allows having a reliable foundation for developing cross-platform applications with a highly interactive interface.

Code Analysis Technique

Moreover, AST-based analysis is selected as a basis for analyzing the code to ensure proper recognition of code structure.

Architectural Design

As far as the architecture of the project is concerned, it features modularity which ensures scalability and maintainability throughout the whole development process.

Resource Availability

Finally, all required libraries and frameworks are widely recognized and documented.

3.8.9 Operational Feasibility

Integration of the software into the workflow of the developer and tester becomes easy as everything can be done in a single environment. Since the software runs on the desktop, users do not need to rely on web-based applications.

Due to the ability of the software to conduct automated tests, less manual work is required, thus making work more productive. Introduction of an artificial intelligence interface makes the software more user-friendly as it makes many tasks easier to complete.

Overall, this software is very helpful in speeding up the process of testing software applications. Therefore, the software would be applicable to both research, academic, and business purposes.

Chapter 4

Software Design Specification

Chapter 4

Software Design Specification

4.1 Purpose

The Software Design Specification provides the technical background for the organization of the Unit and Integration Testing Copilot System, its internal structure, and the main operations that are performed in order to accomplish the testing procedures of the solution.

This chapter presents the design perspective of the solution based on the requirements outlined above. The focus will be put on describing the basic layers, key processors, data processing mechanisms, as well as the interaction between the interface and backend processes of the system. This Software Design Specification aims to serve the construction process of the solution while being also used as an academic guide for further analysis of the system.

4.1.1 System Objectives

The design of the system is guided by the following objectives:

- Improve quality of the test
  The proposed approach will aim at the creation of relevant and correctly structured tests, including both unit testing and integration testing. By analyzing the code of a program, the system will be able to find critical execution paths, potential dependencies and error-prone sections.

- Increase efficiency of development process
  Since the system aims at automating analysis, testing, and results rep-

resentation, the developer will have fewer opportunities to interfere in these tasks, allowing him to devote more attention to coding.

- Reduce laabor-intensive aspects
  Proposed architecture of the software solution decreases necessity in writing tests manually, which proves valuable especially at the initial phase of project implementation.

- Provide complete functionality of test flow
  This system will perform all steps of test generation, from analysis of code, through generation, execution of tests up to their results presentation.

- Ensure flexibility and maintainability
  The fact that the system is divided into modules enables us to change it at any moment, updating and adding testing methods, frameworks, or improving artificial intelligence integrated into the software.

- Traceability assurance
  A good database structure will ensure traceability of connections between source code files, results of analysis, test generation, test execution and report.

4.2 Software Architecture

Unit and Integration Testing Copilot System incorporates a three-tier architecture model to achieve proper distinction between the process of interacting with users, processing business logic, and managing data. This architectural structure enables better system modularity, scalability, and evaluation.

Following is the list of layers used in the system's architecture:

- Presentation Tier (Client Tier)
- Application Tier (Business Logic Tier)
- Data Tier

4.3 Presentation Layer (Client Tier)

Presentation Layer handles all the actions performed by the user. This layer includes a user-friendly interface, through which one may conduct all test functions, select test projects and operate them, as well as analyze the results received from running the test procedures.

The user interface is implemented with the help of Angular and Electron software, and hence, we have both modern approach in terms of design as well as functionality that is typical only for desktop application.

4.3.1 Interface Components

Presentation Layer includes several user interfaces, depending on various aspects of the process of testing

Authentication Interface

The authentication layer takes care about the authentication of the user in the system, presenting appropriate log-in window and logging in the user. The main task of the layer is to maintain the link between user preferences/workspace and the account of the user.

Dashboard

The dashboard layer represents the landing page of our system and gives an overview of activities taking place currently. Also, it represents a tool for navigation to other layers of our system and overview of the last actions made by the user as well as access to all settings. The following user interfaces belong to the unit testing workspace:

Unit Testing Workspace

This is the space where the development of all the problems related to the generation and monitoring of unit testing can be addressed. You could find some interfaces like the following to

- File Explorer
  It is from here that you select the project folders and then pick out the source files to run the automatic unit test process on.

- Code Editor/View Panel
  Here is where you can see the source code and analyze what is in it, as well as obtain the context of testing.
- Test Generation Panel
  This is where you access the software needed to create the automatic unit test cases.
- Test Suite Panel
  Once the test cases are created, they appear here.
- Coverage Panel
  This is where you get coverage data such as summary reports for the quality of the testing process.
- Execution Terminal Panel
  Here is where you will get all your output data from your software operations.

Integration Testing Workspace

The aim of the workspace is to provide an opportunity for carrying out integration testing when you need to check how different components or modules interact with each other. You will be able to access all required controls to control the process of integration testing and analyze its outcomes.

Profile and Settings Module

Settings and Configuration of the Module With the module, you will be able to handle your configuration settings and adjust your interface environment. The module will facilitate the interaction.

4.3.2 Responsibilities of the Presentation Layer

Presentation layer functions are:

- It gets input from the user in the form of project path, file, and testing requirements.
- It provides you with the necessary means to develop, edit, run, and analyze the results of your tests.

- It displays the test cases developed, output generated while running the test, and any warning or error messages that occurred.
- It presents the testing results in an understandable format.
- It communicates instructions for the back-end operations using structured IPC messages.
- It lets you know the status of various processes operating within the system.

4.4 Application Layer (Business Logic Tier)

The Application Layer plays a vital role in carrying out the processing operations in the application. It is here that activities such as analysis of codes, creation of test scenarios, carrying out of tests, and evaluation of tests happen. Additionally, this layer manages the communications between all layers.

Essentially, it is the core of the application that ensures all testing operations within the application are done efficiently.

4.4.1 IPC Orchestration Layer

The IPC (inter-process communication) orchestration layer acts as the controller of all communications between the Angular interface and the Electron processes. As both processes occur separately, it is necessary to have this communication layer.

This ensures that any action taken within the user interface translates to actions being taken in the back-end side and feeding back the outputs to the interface effectively.

4.4.2 Code Analysis Engine

Code analysis engine would analyze the code chosen by the user and extract information regarding the structural elements of the code and its quality properties. This functionality is crucial for the successful realization of automatic test generating process due to the fact that it requires knowledge about the structure of the code.

In order to provide information for further steps, the following actions should be done:

- Recognition of class, functions, and methods – detection of primary elements of the code;
- Detection of imports and dependencies – detection of all the elements that are used by the code;
- Detection of modules – determination of relationships between the modules of the application;
- Calculation of metrics – determination of complexity of the code;
- Detection of the highest priority elements – definition of the code elements which deserve special attention when testing is performed.

The outcome data would be passed to further stages such as generation engine and other modules.

4.4.3 AI-Assisted Test Generation Engine

Test case generation will create the unit tests and integration tests based on the structure of the analyzed code. This module takes advantage of the information found about the source code when analyzing it, in order to find intelligent ways of creating test cases appropriate to your environment.

During the process of performing unit tests, this module will analyze each function, method and class individually. In contrast, the creation of the integration test cases will pay attention to the manner in which different modules work together, including services calls and data flows through different paths of execution.

Basically, this module acts as a crucial part in all of the process since it allows using the findings of the analysis in order to proceed with testing.

4.4.4 Test Execution Coordinator

Module for Test Execution Coordination

The test execution coordination module is responsible for executing the tests generated and collecting the results of the tests from the testing frameworks. This module is used as an intermediary between your test files and your code's testing process.

The major functions performed by this module include:

- Preparing the commands for execution: The module is used to prepare all the necessary commands in order to start the test execution process.
- Execution of test scripts: The module is responsible for the execution of the test scripts.
- Collecting the outputs and the log files: The module is used to collect all outputs and the log files which are created in the course of testing.
- Determining the successes and failures: This module determines whether or not the tests were successfully executed.
- Detecting exceptions and other execution problems: This module can detect any exception or problem that occurs during the test process.

Using this module, test cases do not stay unexecuted in files. Instead, they are executed.

4.4.5 Coverage and Quality Analysis Module

It is the responsibility of the coverage and quality assessment module to perform an analysis of the effectiveness of the testing process carried out. One can easily identify which section of the source code has been tested and how well it was able to detect the error.

Some typical outputs of this module include:

- Coverage statistics: overall data on the code coverage;
- File or module level reporting: data on the coverage of specific files or modules;

- Quality evaluation: metrics that determine how thoroughly your test set covers the program;
- Mutation evaluation: the metric that determines how effectively your test set detects errors or mutants.

Such an output will help determine whether the generated test cases are adequate and whether your project has been adequately tested.

4.4.6 Validation and Repository Management

Before the execution stage, there will be a process of validating the generated artifacts to make sure they have been successfully developed. During the validation process, there will be a checking process performed for the structure, arrangement, and consistency of the test artifacts before they can be saved and executed.

Under the storage management process, the process involved includes arranging the generated tests and outcomes. Through this process, future testing becomes very easy.

4.4.7 Reporting Module

Reporting Module – The reporting module has the responsibility of displaying the summarized information on the execution of test cases. Data presented in the form of reports will be easier to analyze than that presented in the form of logs.

The following are the components of the Reporting Module:

- Test Execution Summary: Presents an overview of the entire test execution.
- Pass/Fail Information: Gives statistical information regarding the pass/fail information of different test cases.
- Code Coverage Information: Gives the details of the parts of the code that have been tested during the testing phase.

- Error Information: Explains the nature of any error during test execution.
- Past Results View: Enables one to analyze past test results in order to determine progress.

Through this process, the data generated by the backend becomes easy to interpret by the users.

4.4.8 Responsibilities of the Application Layer

The duties of the Application Layer involve:

- Managing the entire workflow: This involves managing the entire workflow from receiving the source code until the creation of the reports.
- Interfacing between the modules: The Application Layer functions as the mediator module to guarantee the proper functioning of the other modules such as analysis, generation, execution, and reporting.
- Verification of the test cases: This involves verifying that the test cases are valid prior to executing them.
- Workflow rule management: The Application Layer manages the entire workflow rules and also manages the exceptions in the workflow.
- Making the module scalable: This is achieved through modularity.

4.5 Data Layer

Data Layer supplies all information necessary for the process of analyzing, creating testing, executing it and storing its outcomes. That is, Data Layer guarantees preservation of all necessary information required for future examination.

It allows both temporal and permanent provision of information and is extremely important for reproducibility and traceability.

4.5.1 Local Workspace Files

The basic source of the input information is defined as the selected project directory created by the user himself. Files and directories including source code for further analysis and testing are considered.

Such files are analyzed and further processed during the process of generating tests.

4.5.2 Generated Test Artifacts

All unit and integration tests can be treated as system artifacts which should be saved for future examination and testing purposes.

Saving of the output is important both for work of developers and demonstration of system capabilities at university.

4.5.3 Execution Results and Logs

It keeps all the data related to the generation process, such as logs, console outputs, error descriptions, and states of tests being passed or failed. Speaking of the debugging process, this evidence is very helpful when creating a report about what the system was doing.

4.5.4 Coverage and Quality Snapshots

The same applies to the metrics of your test runs; they may be saved as snapshots and compared with each other to identify improvements in your skills.

4.5.5 User Preferences and Settings

All the individual settings made by you are saved in the Data Layer as well. This aspect makes the usage of the system even more convenient because you will not have to make the same settings again after restarting the application.

4.5.6 Optional External Sources

Additionally, the possibility of communicating with external sources, which might assist with further analysis and test creation, might occur. However,

it is just an extra feature, and there is no need to use external AI models or packages to work with our system.

4.5.7 Responsibilities of the Data Layer

The Data Layer is designed to do the following:

- Saving of analysis and tests: The data layer ensures the storage of the results of the analysis and tests created in the system.
- History tracking: The layer helps you track your history, as well as the results of your past tests so you could check on them whenever you needed to.
- Saving quality data: With this function, the coverage and quality data can be saved for analysis.
- Traceability: The layer offers traceability, going from source code to testing results.
- Reproducibility: This allows you to run the exact same test again, which might come in handy if you need to debug something or provide academic evidence.
- Growing: It's easy to migrate the data layer into the cloud in the future if needed.

4.6 Data Flow

This design allows for a uniform way of handling input and output operations while testing.

Firstly, you will have to submit some details concerning the projects and file paths on your computer using a GUI interface. Then, the gathered information will be sent to the backend using the IPC technologies.

4.6.1 Input Stage

Firstly, you will have to submit some details concerning the projects and file paths on your computer using a GUI interface. Then, the gathered information will be sent to the backend using the IPC technologies.

4.6.2 Analysis Stage

Further, the Application layer starts code analysis process. It includes the analysis of the structural code, calculation of metrics and preparation of the necessary background for test creation.

4.6.3 Test Generation Stage

On the base of gathered data, test cases will be generated and checked for errors.

4.6.4 Execution Stage

Here, the system will generate logs and messages concerning status and outputs of the test execution.

4.6.5 Evaluation Stage

Then, the valid test cases will be run according to the described in the test execution coordinator algorithm.

4.6.6 Output and Storage Stage

Finally, the data received by the system and its analysis will be made available for users through the GUI interface as well as stored within the Data Layer.

4.7 Detailed View

4.7.1 Logical View

Class Diagram

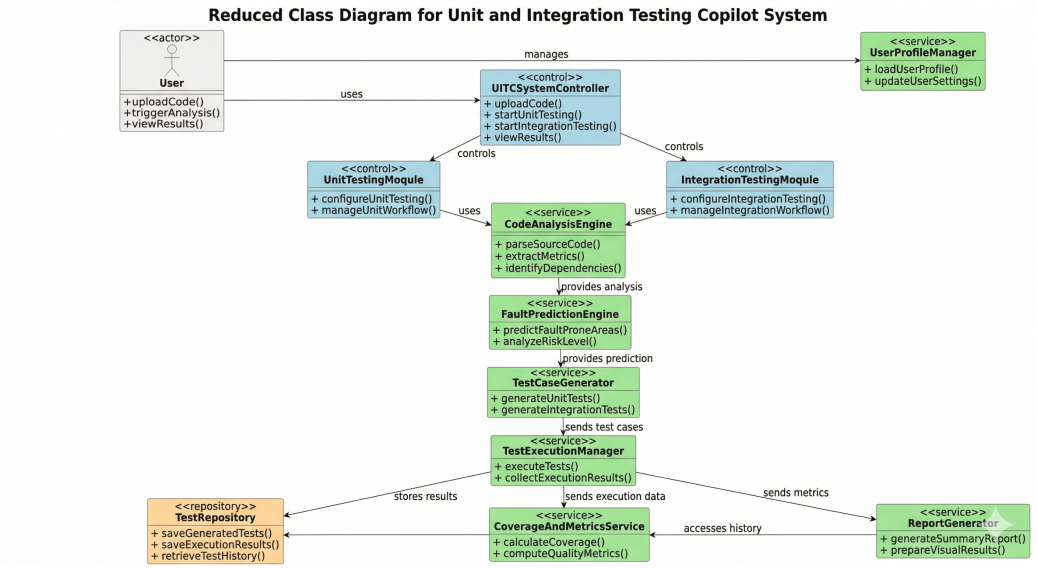


Figure 4.1: Class Diagram

4.8 Dynamic View

4.8.1 State Machine Diagram

UITC Main Flow State Machine

UITC Flow State Machine is responsible for all actions during navigation within the Unit and Integration Testing Copilot System. The main steps of this process include: login to dashboard; transition from dashboard either to unit testing flow or to integration testing flow; from one testing flow to another or to logout, then returning to login page.

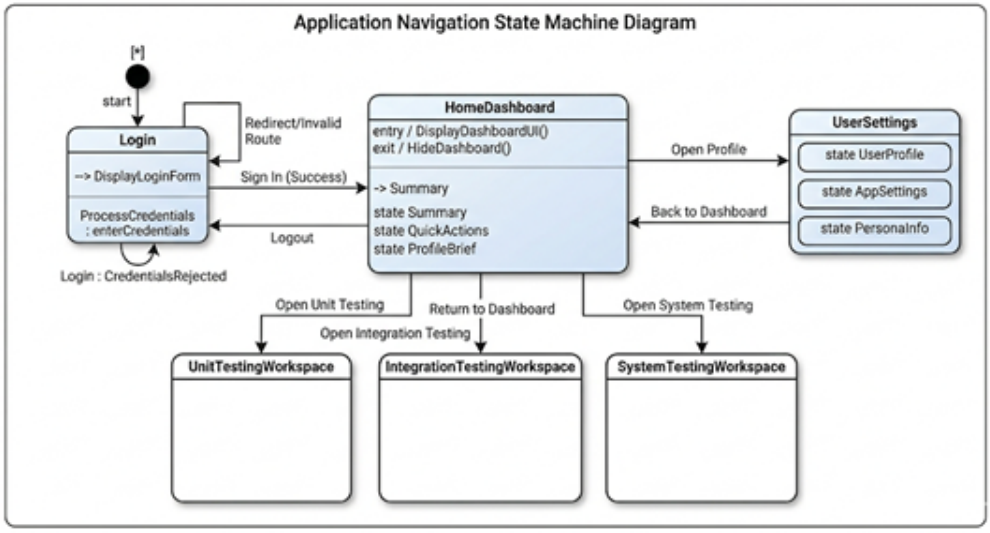


Figure 4.2: UITC Main Flow State Machine

Unit Testing Flow State Machine

Unit testing flow state machine defines the sequence of actions performed while executing a unit testing process. The unit testing process involves such actions as loading/choosing the code, its analysis, creating unit tests, executing them and checking the results. Having analyzed the results, the user is ready to execute another unit testing process.

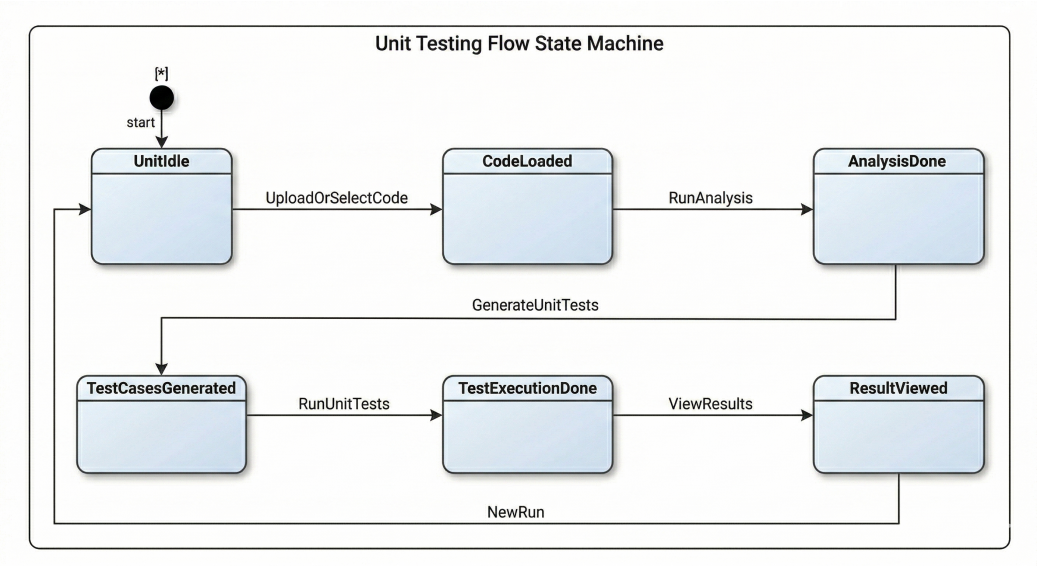


Figure 4.3: Unit Testing Flow State Machine

Integration Testing Flow State Machine

Integration testing flow state machine is responsible for the testing module behavior while performing an integration testing process. These actions include: choosing the modules to be tested, analyzing dependencies between them, creating integration tests, executing them, and analyzing the received results. Having done so, the user may start another cycle.

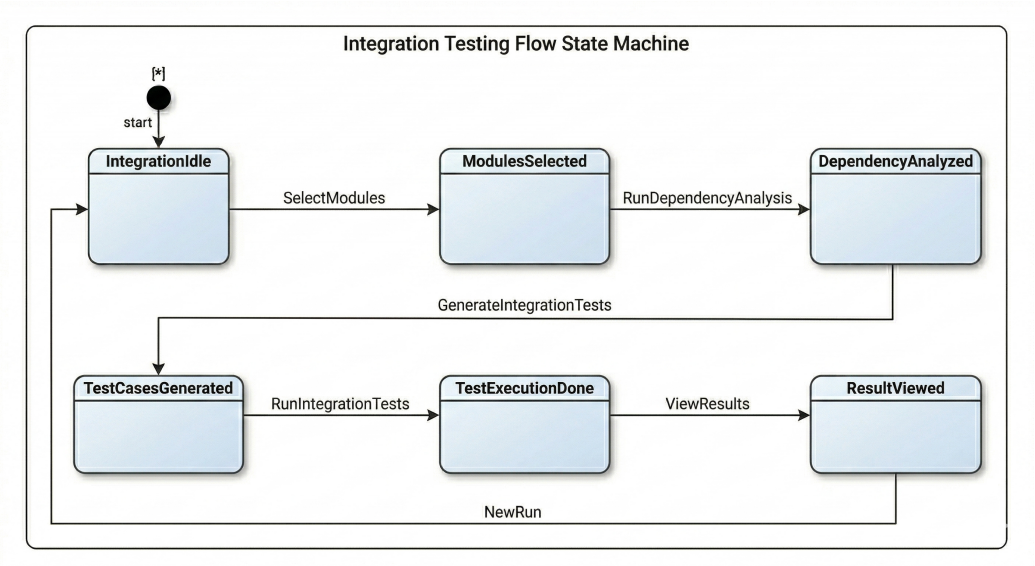


Figure 4.4: Integration Testing Flow State Machine

4.8.2 Sequence Diagrams

UITC Main Flow Sequence Diagram

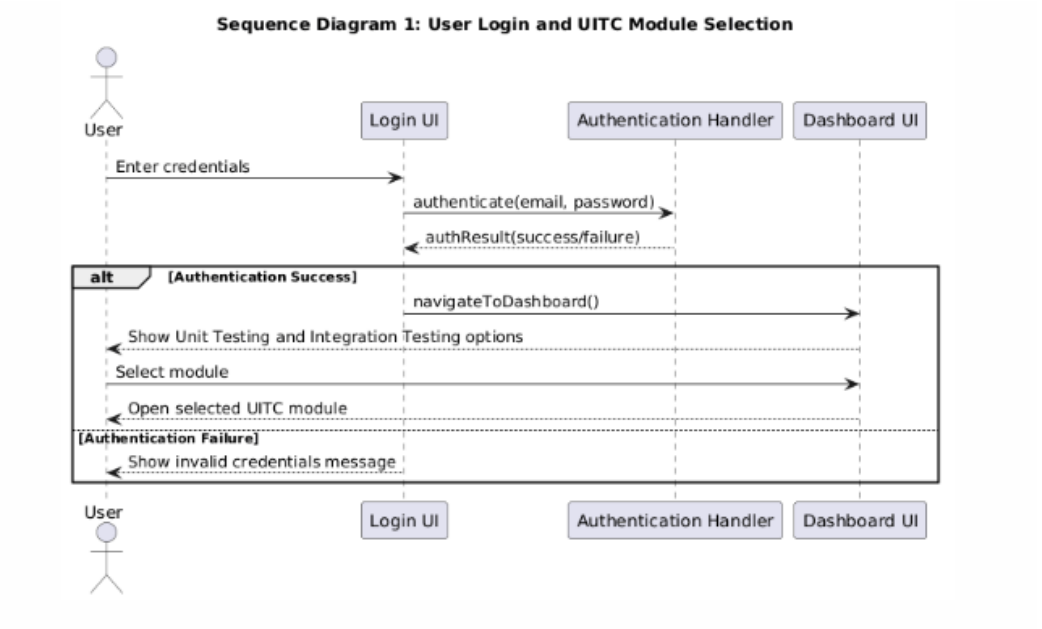


Figure 4.5: UITC Main Flow Sequence Diagram

Unit Testing Flow Sequence Diagram

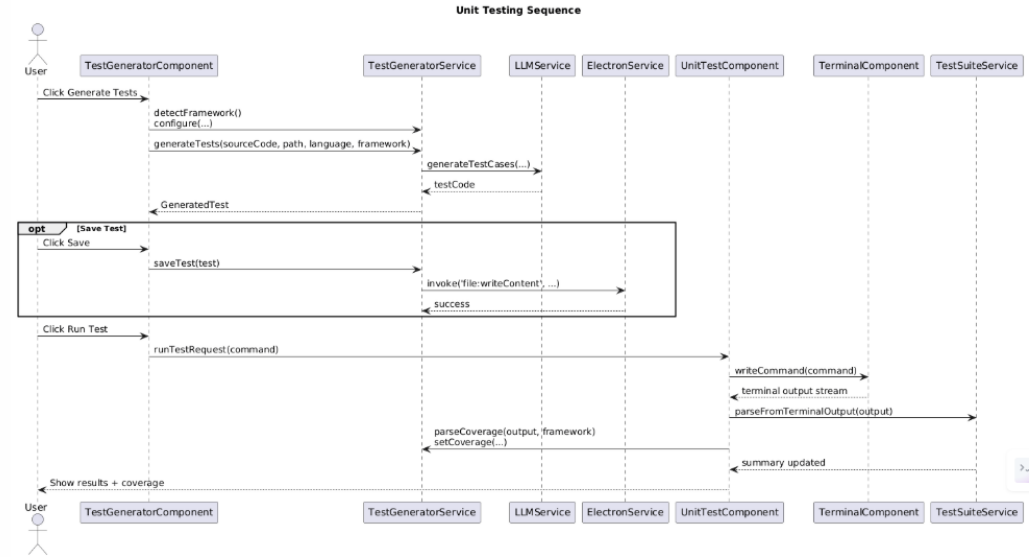


Figure 4.6: Unit Testing Flow Sequence Diagram

Integration Testing Flow Sequence Diagram

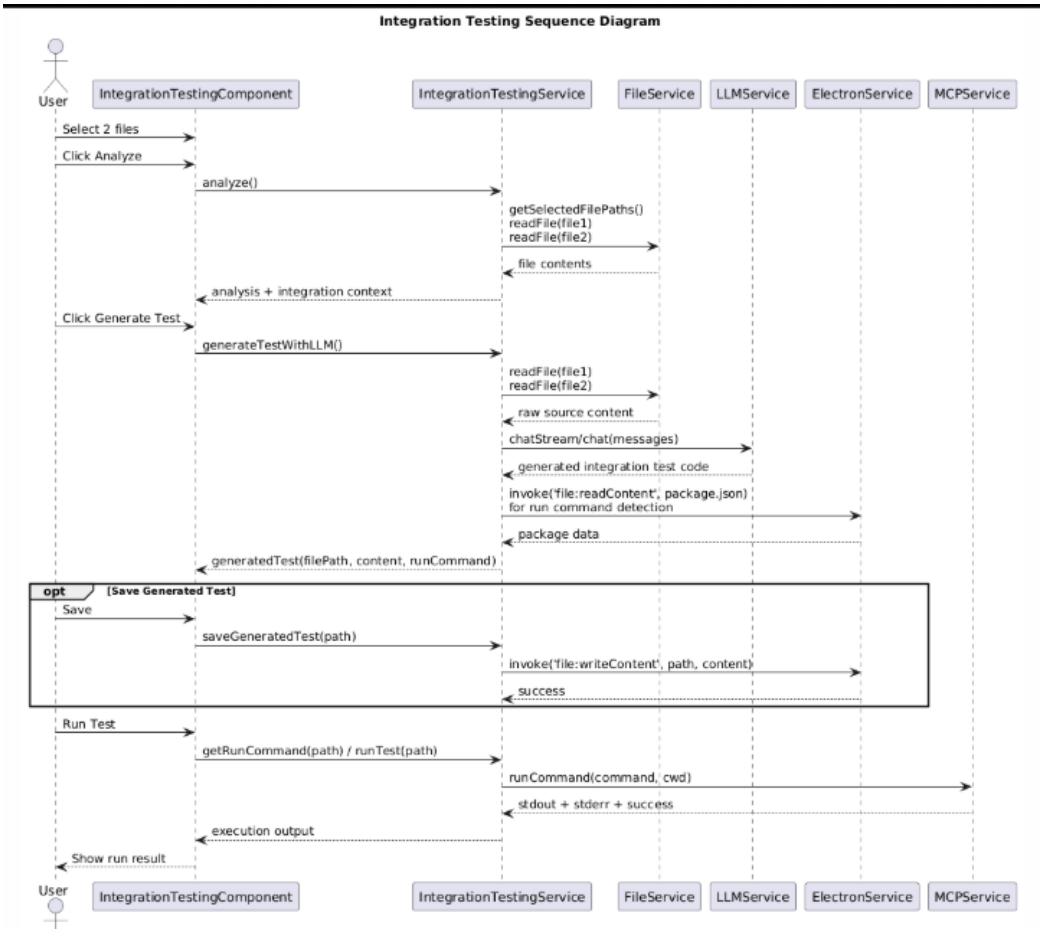


Figure 4.7: Integration Testing Flow Sequence Diagram

4.8.3 Deployment View

The deployment diagram shows the deployment architecture of the Unit and Integration Testing Copilot System within the developer's workstation. This shows the electron application together with the runtime processes alongside other components such as the workspace storage component. Besides, it shows the system interactions with the large language model API to perform some complicated actions.

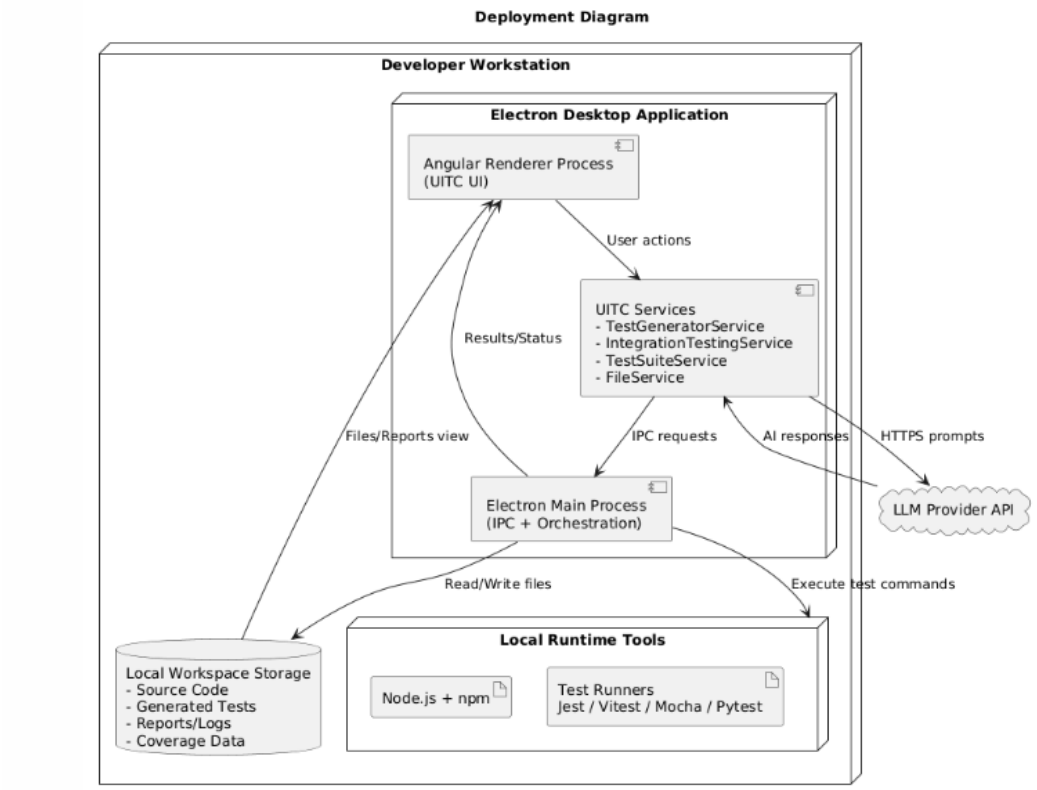


Figure 4.8: Deployment Diagram

4.8.4 Data Model Diagram

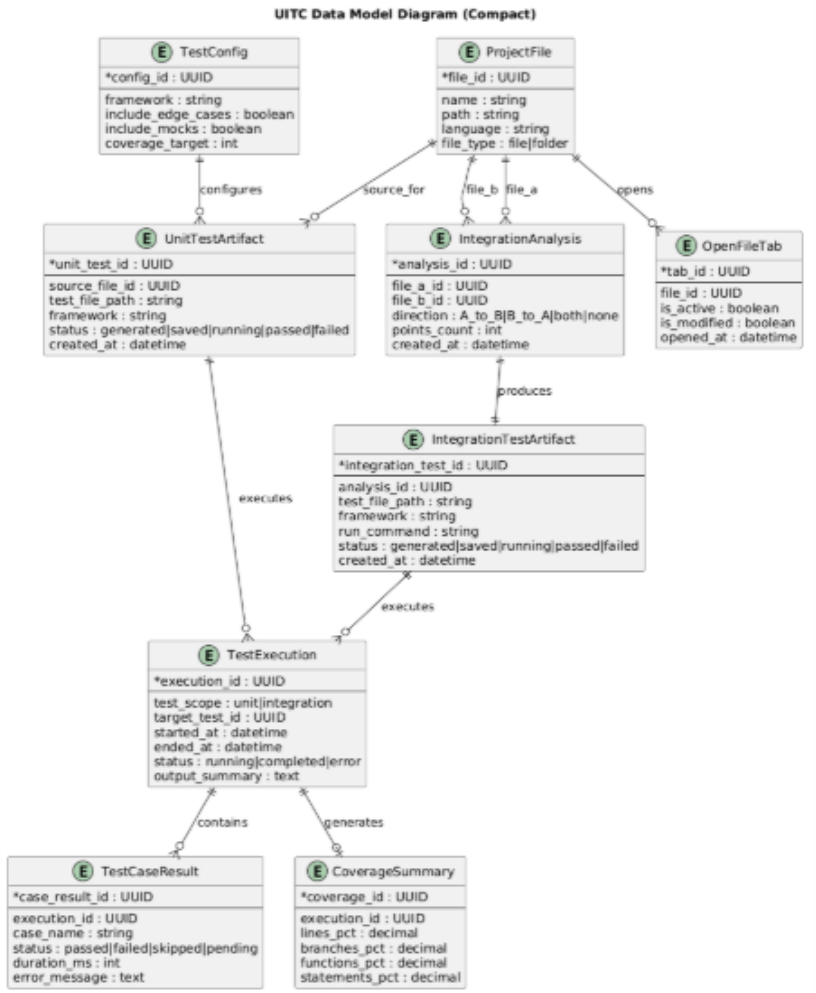


Figure 4.9: Data Model Diagram

4.8.5 Wireframes

Login Wireframe

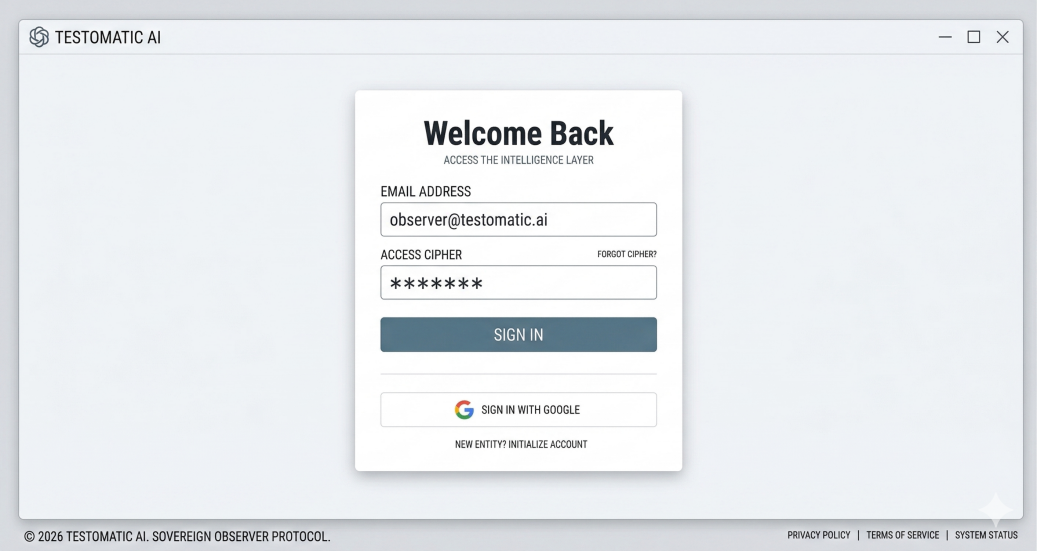


Figure 4.10: Login

Main Dashboard Wireframe

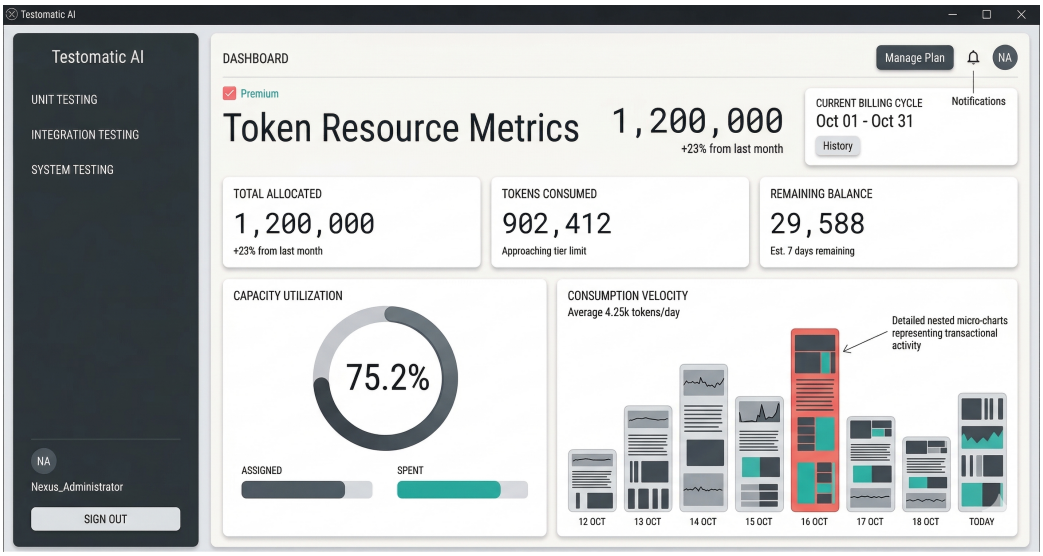


Figure 4.11: Main Dashboard

Testing Workspace Wireframe

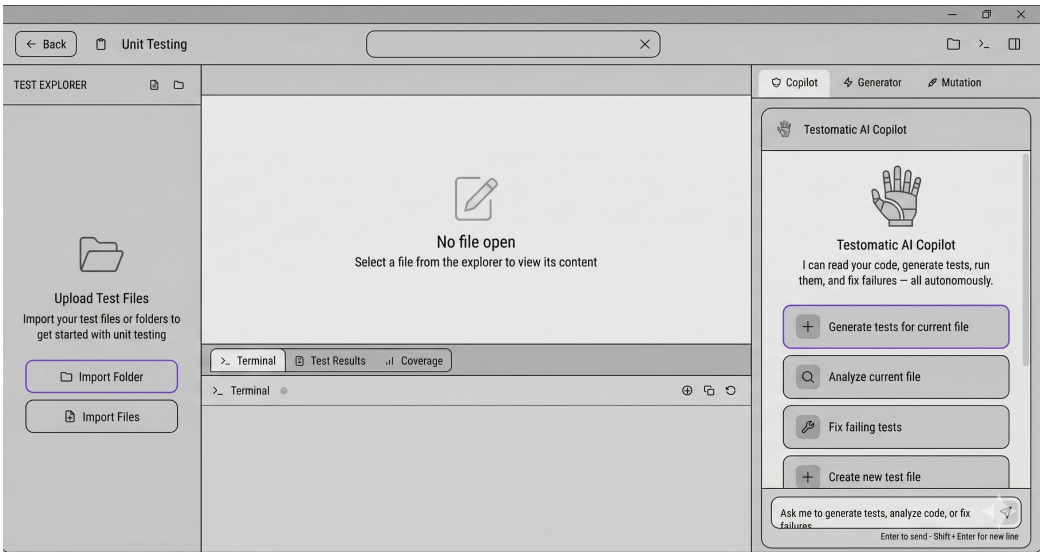


Figure 4.12: Testing WorkSpace

Chapter 5

System Implementation

Chapter 5

System Implementation

5.1 Implementation Overview

5.1.1 Objectives of Implementation

At this stage, the idea for the desktop application has become a real functioning software solution. What has been considered important in the process of development is providing an effective means of conducting testing processes, from choosing the codebase to its visualization.

The issue of modularity has been also taken seriously. It has been deemed vital to have the developed system split into the necessary modules that will be responsible for the respective activities related to visualization, management of logic, and performing actions, thus ensuring that the modification of the corresponding feature would not affect the whole system. This objective has been achieved through the use of the component-service architecture, in accordance with which we have distinguished the control interface and multiple specific services that have the responsibility of performing certain functions such as code analyzing, generating tests, and executing commands and visualizations.

In addition, the aspect of scalability has been taken into consideration. We have developed a very modular implementation based on a data model and proper management of commands and services that will allow the possibility of incorporating third-party APIs.

5.1.2 Implementation Scope

Implementation includes the following five interfaces:

- Login and navigation to the application entrance.
- Dashboard - Interface which enables selection of the modules and provides an overview of your work.
- Unit Testing environment - Interface that allows creation of tests and analysis of their outcomes on a particular file.
- Integration Testing environment - Interface where interaction between files and modules is tested.
- Settings - Interface responsible for managing your user account settings.

Within this approach, all required operations for unit and integration testing (file-based tests creation and execution) will be executed. Currently, there is no authentication on a server level but only based on navigation. Settings of the user profile are adjusted in a manner that is prepared to be linked to APIs.

In summary, the application contains all required functionality for demonstration purposes, as well as future professional usage.

5.1.3 Technology Stack Overview

This system has been built using the desktop-based architectural approach. Listed below are some of the essential technologies utilized for building this system:

- Angular: Utilized in building the user interfaces and navigating the application.
- TypeScript: Utilized for the typing capabilities of the application along with good structure in organizing the business logic of the application.
- SCSS: Utilized for styling the application interfaces as well as making sure that those interfaces are responsive.

- Electron: Utilized for running the system as a desktop-based application.
- Inter-Process Communication (IPC): Utilized to facilitate communication between the visual interface and the underlying processes of the system.
- Service-oriented architecture: The service-based approach is utilized to organize the business logic associated with analyzing information, performing tests generation, execution, and results processing.
- LLM integration: Utilized for connecting to LLMs and generating tests.
- Dynamic test execution: Utilized for executing dynamic tests using various testing frameworks.

These are the technologies that form the basis of this system architecture.

5.2 User Interface Implementation

5.2.1 Login and Authentication Interface

The log-in module is designed as a one-way portal into which users log in. The actions that the log-in module performs include log-in and federated log-in.

At this moment, both these actions simply redirect to the dashboard page by client-side navigation. Such simplicity guarantees that all features of the system become accessible regardless of having no need for any sort of back-end validation process implementation.

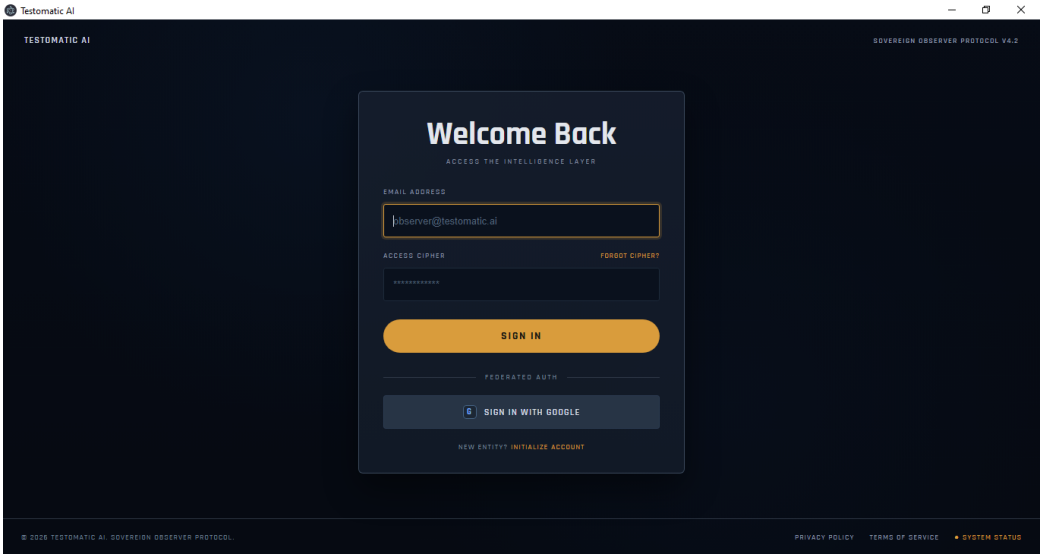


Figure 5.1: Authentication Interface

5.2.2 Dashboard Interface

The Dashboard will be used as the main means of moving around within the application. In this regard, the user will gain access to Unit Testing and Integration Testing through the dashboard, which also contains the measures of the system's performance in the form of metrics displayed in chart form.

The creation of the dashboard will involve structured data modeling that makes its usage convenient and flexible. Thus, incorporating new data will become easy. Furthermore, it is worth noting that the dashboard will be responsive and hence suitable for various screen sizes.

Figure 5.2: Dashboard Interface

5.2.3 Unit Testing Workspace Interface

The Unit Testing workspace is supposed to aid in every step of the entire unit testing process. The Unit Testing workspace is composed of several functional modules like navigating through files, browsing through the code, creating tests, executing them, and measuring coverage.

The first step would be selecting any of the source files for unit testing. Then there will be a step where you create tests from the code that you have selected. Execution of these tests will be performed using the command processor, and output parsing will be done to get test results and coverage information.

Output parsing is necessary for the correct functioning of this program.

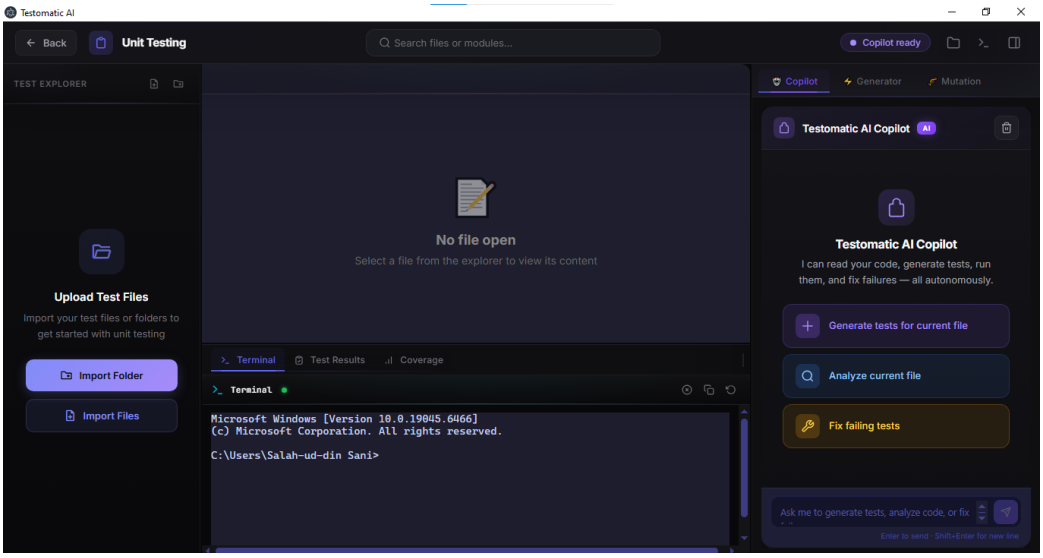


Figure 5.3: Unit Testing Interface

5.2.4 Integration Testing Workspace Interface

The Integration Testing workspace focuses on integration testing among more than one code units. This includes choosing two code files and studying the interaction between them.

These tests are then logically generated. Execution instructions are then automatically generated by the software based on the development environment at hand, hence making it easy to conduct integration testing without any user configuration.

Also included in the user interface is the progress status indicator that enables users to track different phases of the test, including analysis, execution, and result generation.

5.2.5 User Settings Interface

the User Settings interface allows you to view and update your profile-related information. It includes fields for account details, usage data, and configuration settings, along with options to save your changes or go back to the dashboard.

The implementation uses organized state management, which allows the current local data model to be easily swapped for API-driven data later

on. This ensures a smooth move toward fully integrated user management without needing to change how the interface is built.

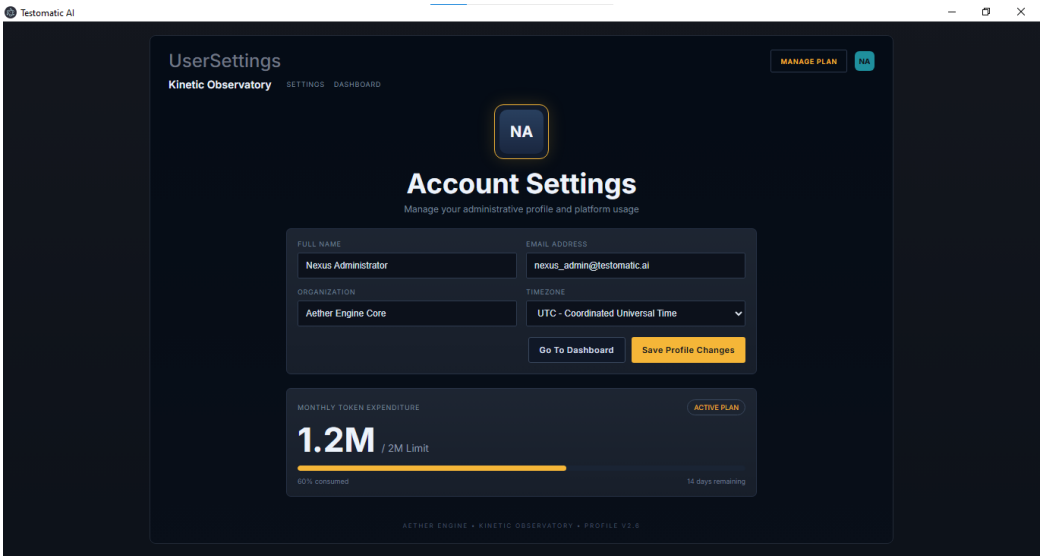


Figure 5.4: User Settings

5.3 Unit Testing Module Implementation

Unit testing module consists of an orchestrated pipeline including a page level orchestrator, generator services, execution services, and terminal interaction. The controller is responsible for handling users' inputs regarding interaction with the user interface, while separate services are responsible for code generation, execution, and handling results.

5.3.1 Code Selection and Loading Mechanism

Selection of files is performed by the file service, which imports individual files and folders. Folder selection determines the working directory that will be used in automatic framework identification and in command execution generation.

Load operation is performed using stateful approach, where a tab-based approach is employed. Files are loaded into open tabs, and no new tabs are created; instead, the open tab is used. Currently open file is used as the

input in generating tests, so the test is processed according to the selection made by the user.

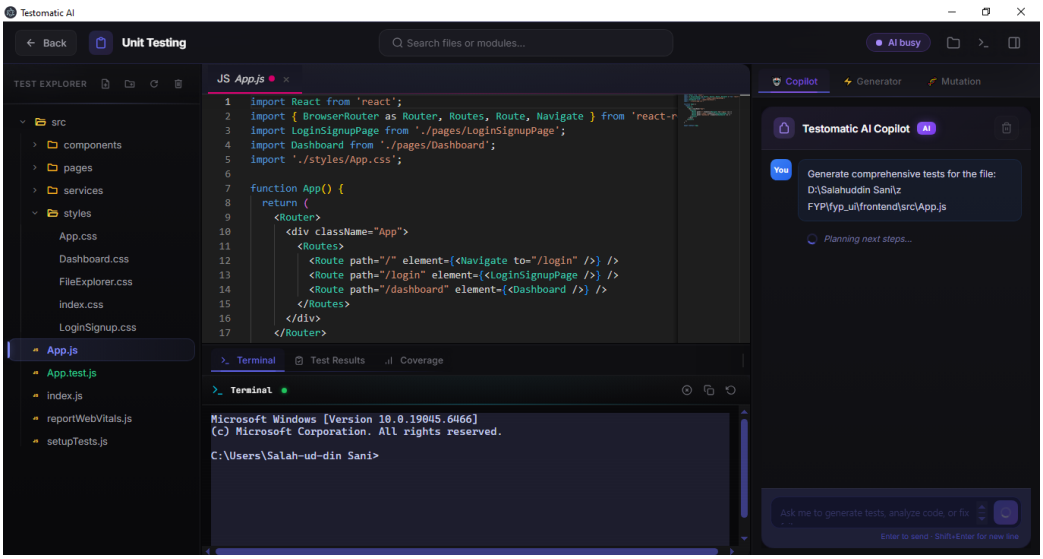


Figure 5.5: Load CodeBase

5.3.2 Code Analysis Implementation

Lightweight operations and efficiency are key characteristics of the analysis layer. Detection of languages is accomplished by identifying file extensions, while simple statistical metrics such as line counts are used for context.

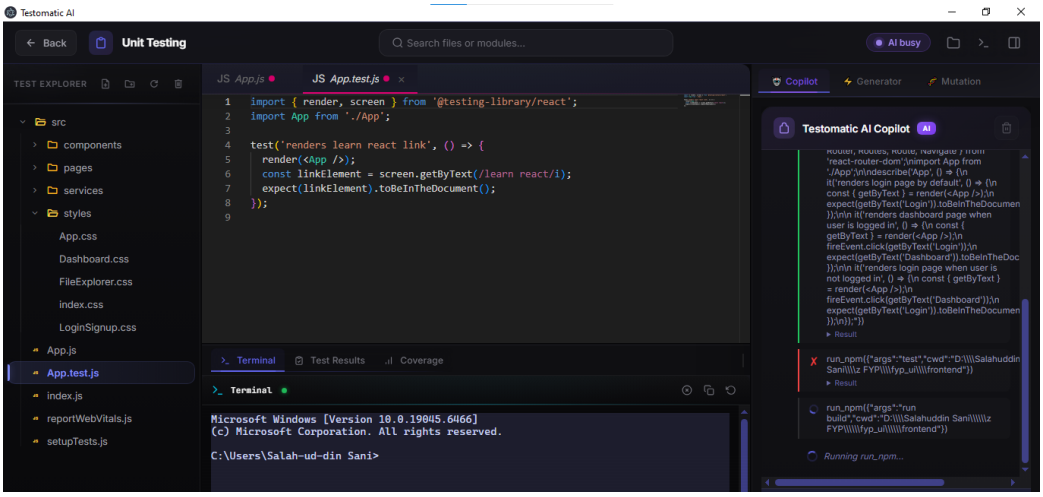


Figure 5.6: Code Analyzing

5.3.3 Unit Test Case Generation Logic

Test generation takes place via a flexible pipeline approach. Test requests are generated in a standard way, while artificial intelligence services play an important role in generating test code.

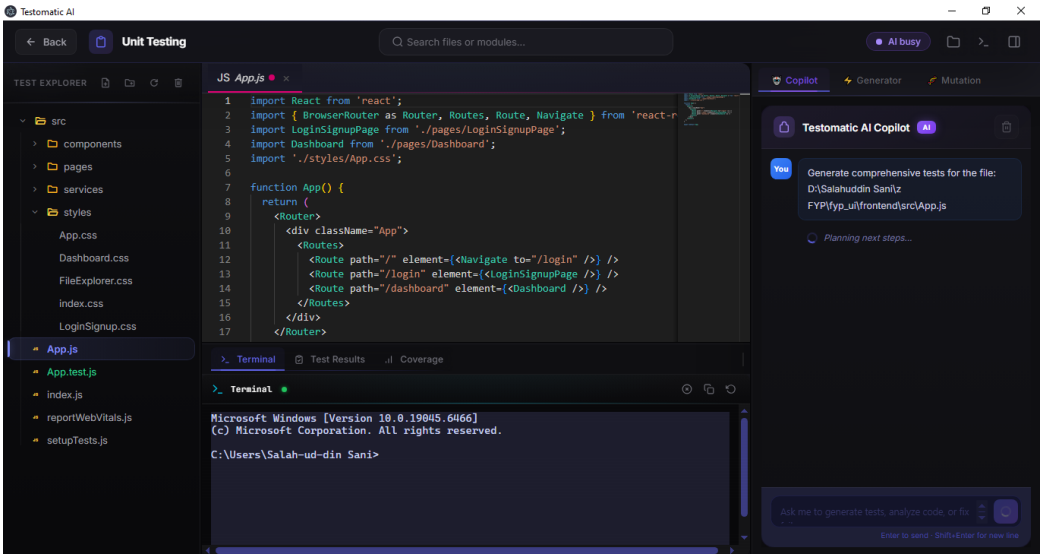


Figure 5.7: Unit Test Cases Generation

5.3.4 Test Execution Process

The test execution phase includes command generation and result evaluation. The results generated through test execution are evaluated for instant feedback as well as summaries.

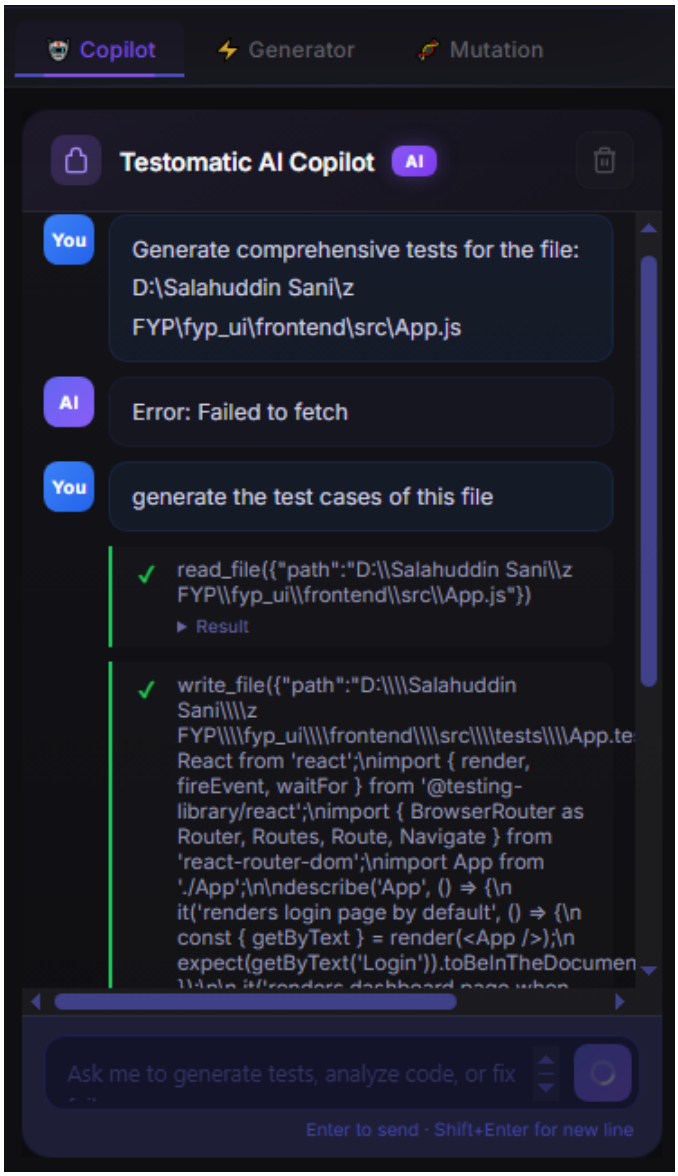


Figure 5.8: Test Execution

5.3.5 Coverage and Metrics Calculation

The results that are generated using the test execution phase can help in calculating the percentage of coverage attained in the testing phase.

5.3.6 Auto-Fix Mechanism Implementation

Iterative character of the auto-fix procedure implies creation and execution of test cases; detection and fixing of problems with them until their proper execution or achievement of certain limit.

5.4 Integration Testing Module Implementation

The Integration Testing Module enhances the capabilities of the tool by not only examining individual source code files but also ensuring the quality of interaction among them when being part of the same program. Contrary to the unit testing approach, this module deals with testing the interaction between different elements in the software application.

5.4.1 Module Selection and Context Building

Firstly, two files are selected, limiting the scope of possible testing actions. This step is required since working with interactions of two files is considered much more straightforward than dealing with several ones. After that, the code of each file is loaded into the memory and all the necessary information is gathered.

Context contains such data as file names, the programming language detected and its complexity.

5.4.2 Dependency Analysis Implementation

This type of analysis can be called context-based interpretation and not a comprehensive static analysis approach. The software can find the dependencies between the chosen modules based on function calls, data flows, and logical dependencies.

This effective approach saves computational resources and provides useful data about the dependencies for tests. At the same time, this solution can be easily extended in case other approaches need to be implemented.

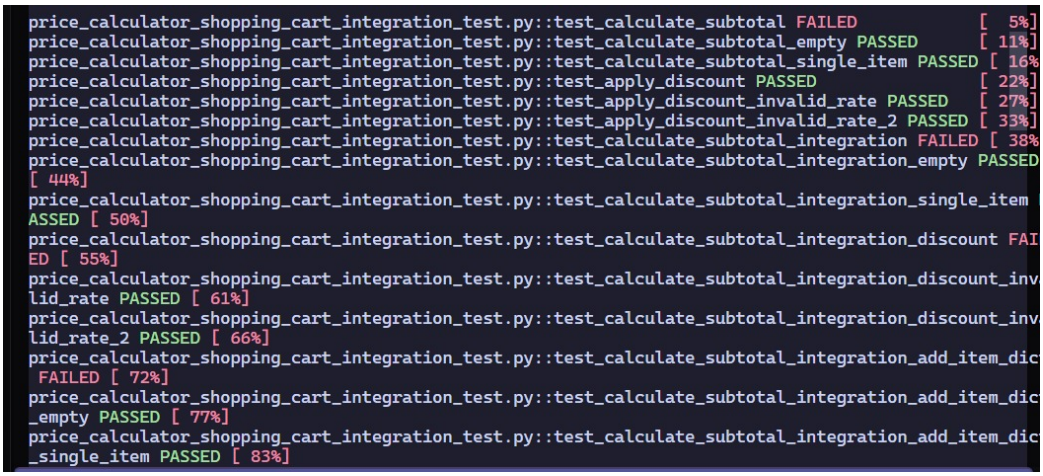


Figure 5.9: Dependencies Downloading

5.4.3 Integration Test Case Generation

The generation of intelligent tests occurs by merging context-specific features from both files being tested into an input structure. Tests scenarios are created that represent real communication flows between components and validate their communication channels and information exchange.

The generation of test cases is facilitated by an AI algorithm that generates test cases based on the programming language being used. The output of the generation process is structured into a standardized output structure that includes test cases, execution instructions, and file names.

5.4.4 Integration Test Execution Workflow

The execution process is designed not to rely on any predefined criteria. The test cases and execution instructions are incorporated into the environment where the former are automatically interpreted based on the framework available in the environment.

The test case execution itself occurs through the invocation of services at

the system level, making it possible to gather outputs and perform verification operations.

5.4.5 Result Aggregation and Reporting

The result of the test for the process will be gathered through the data and internal state of the process. Through such a technique, the results of both successful and unsuccessful tests will be recorded in order to provide a picture of the process of testing.

Although documentation of the process has not been generated yet through the system, there is a systematic organization of data regarding generation, execution, and the results of the process.

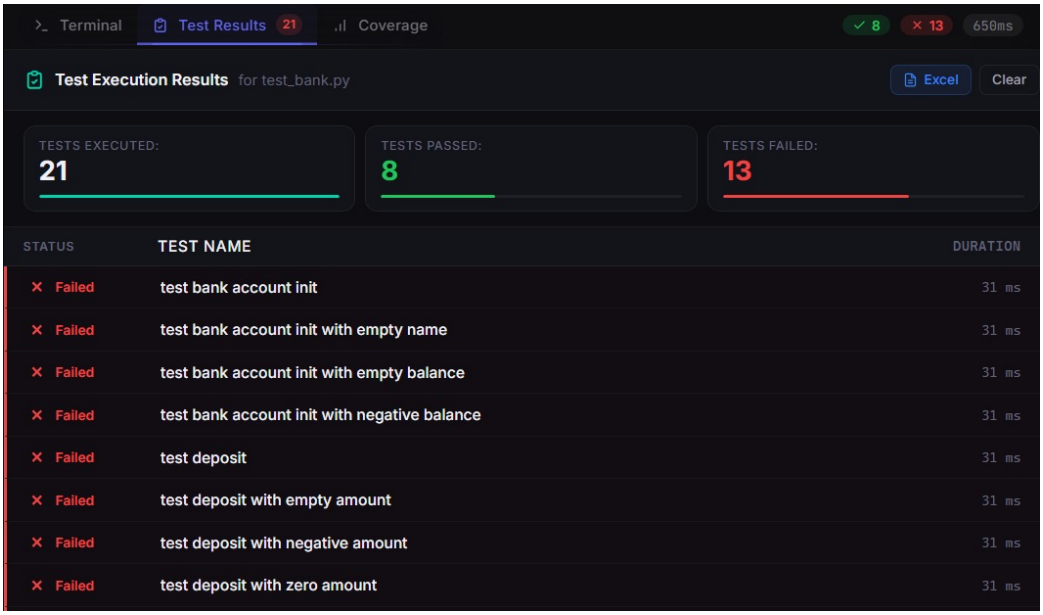


Figure 5.10: Test Results

5.5 Code Analysis and Intelligence Engine

Code Analysis and Intelligence Engine can be regarded as the main component of the decision-making module of this application. It contains numerous analysis engines and mechanisms that enable the application to conduct in-

telligent test generation and execution operations. As for the test execution results, they undergo analysis in the code analysis and intelligence engine.

5.5.1 Static Code Analysis Implementation

Static code analysis is carried out using an extremely light preprocessing mechanism. The language is identified according to the file extension, while some simple metrics like size and code complexity are determined.

The described approach provides fast processing along with sufficient data for analyzing the code. As no heavy parsing takes place, the performance is sufficient to implement this process interactively in a desktop application.

5.5.2 Dependency Extraction Mechanism

Dependencies Extraction is aimed at identifying necessary connections needed to execute and integrate but not at creating the full dependencies graph. The application is able to detect required frameworks and runtimes using information about the project provided and connections between selected modules.

This approach allows maintaining effectiveness of the dependencies management while ensuring necessary accuracy in generation and execution of the tests.

5.5.3 Fault Prediction Logic

The fault management mechanism used in the system is reactive in nature; thus, fault detection is performed depending on the outcomes of the test execution and involves analysis of failures, inconsistencies, and problems with coverage identified within the course of test execution.

The system is unable to predict failures or faults before starting the testing procedure but analyzes all the issues identified during this stage.

Moreover, the application can perform regeneration of tests and modification of the execution policy to ensure proper test coverage and avoid potential problems.

5.5.4 AI-Assisted Test Generation Integration

Finally, another distinctive feature of the current system architecture is the involvement of artificial intelligence in the working process of the system.

To improve performance of the system, the application uses some interfaces allowing it to interact with external models.

Such an approach enables the system to utilize the full capacity of AI systems not only in the process of test generation but also during other activities.

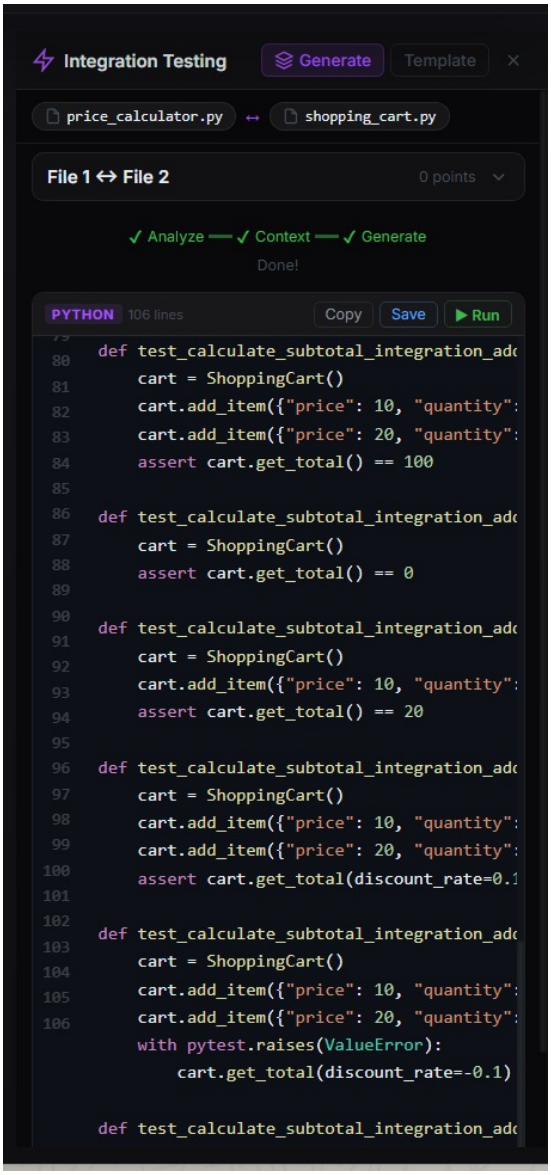


Figure 5.11: Chatbot Responce

5.6 Test Execution and Runtime Handling

5.6.1 Execution Engine Design

The engine itself is an adaptive orchestration layer that encapsulates test execution stages. The engine performs the orchestration of command execution, the execution stage of the command, and result handling without regard to any specific testing framework.

The option of execution command on demand based on the project configuration allows the engine to execute tests in different programming languages within one workflow process. The proposed approach will prevent duplication and reuse the execution process in other testing frameworks.

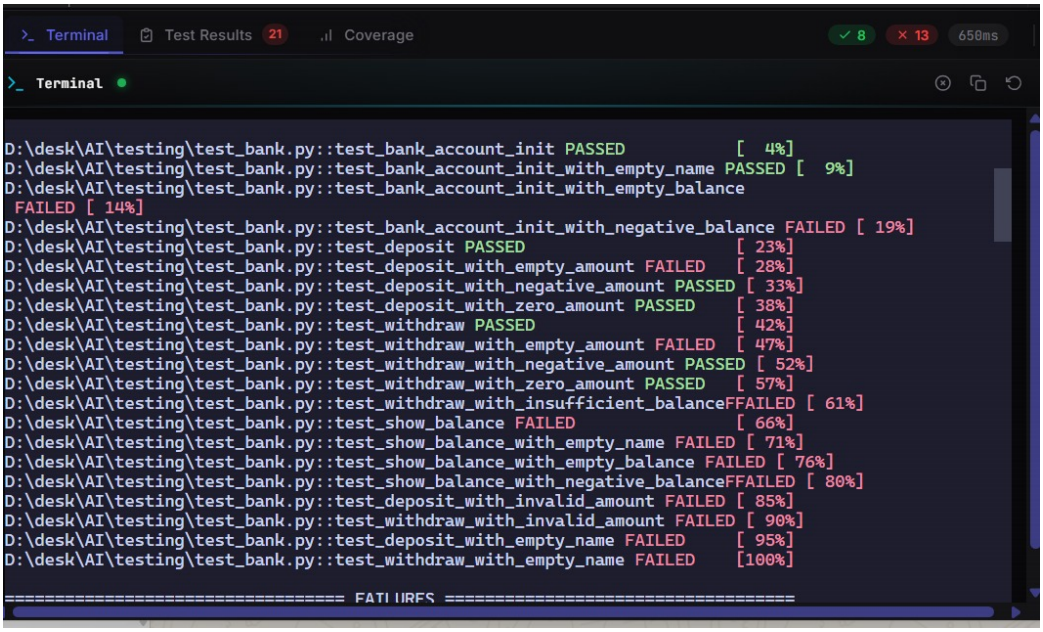


Figure 5.12: Test Case Execution Engine

5.6.2 Terminal and Command Execution Handling

The execution of command is carried out using the integrated terminal-based and service level command execution approach. While performing unit testing, the commands are delivered via an integrated terminal to track their performance live.

During integration testing, the commands are executed through service level processing, offering more formal results and better control over the execution process.

5.6.3 Logging and Output Capture

Logging is modeled as a multi-layered procedure that involves capturing raw logs and transforming them into structured data. The logging mechanism buffers console outputs, processes their format, and identifies signals that signal completion and execution result statuses.

The structured results from parsing are recorded using model-based structures such as execution details, test results, and timestamp information, thus ensuring that both raw and parsed data can be analyzed.

5.6.4 Error Handling and Recovery

Error handling procedures are modeled to enable continued system functioning despite errors. Before executing the program, validation checks are performed to ensure that no faulty execution takes place, such as execution with missing files.

Error recovery procedures include retries, fall back plans, and correction using the auto-fix functionality. Error propagation through structured states enables error reporting in the user interface without disrupting the execution process.

5.7 Coverage and Reporting Implementation

5.7.1 Coverage Data Collection

Code coverage analysis is performed based on data extracted from the test outputs. Frameworks generate command-line arguments unique to each framework and parse them to get the code coverage data.

This way, the program is able to handle different frameworks without having to install other tools.

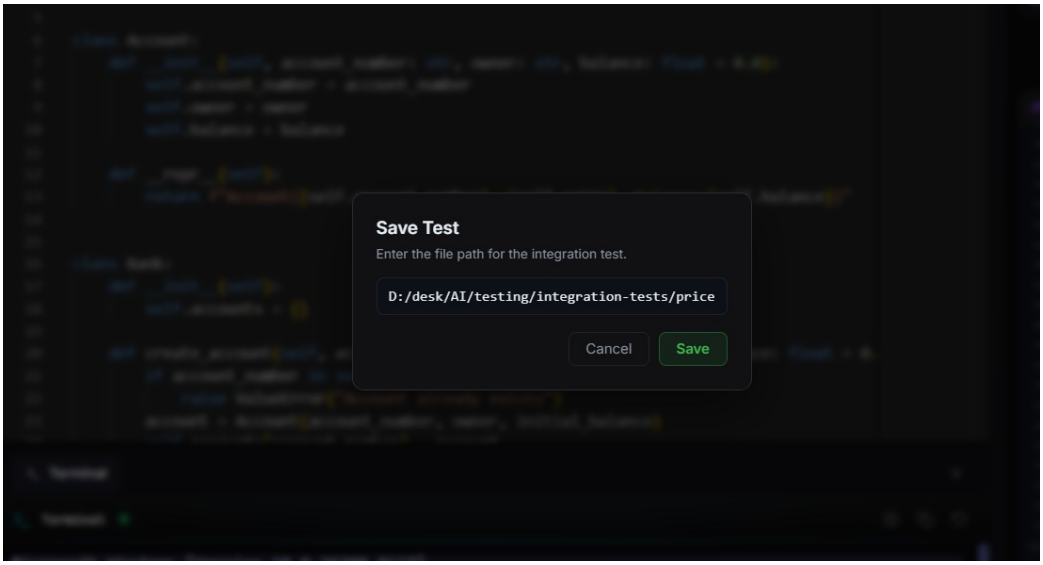


Figure 5.13: Coverage

5.7.2 Metrics Calculation Logic

The indicators will be collected depending on parsed execution results that include execution results, duration of time and coverage ratio. Where insufficient data can be observed, summary data will be considered to make sure that everything is consistent.

To ensure stability, this strategy is chosen independently of diversity of execution results.

5.7.3 Report Generation Mechanism

The creation of runtime summaries is done during reporting based on the combination of execution results, coverage ratio and timestamp. The information will instantly become available to assist future report generations.

5.7.4 Visualization of Results

Execution results are provided via interface panels that display summaries, coverage ratio and execution data. Visualization of execution results is done conveniently.

5.8 Data Management and Storage

5.8.1 File System Interaction

Interactions with the file system occur via controlled operations which help select files, read/write them, and navigate the directory tree. Controlled paths are used by the application to ensure interactions with the file system occur consistently.

5.8.2 Test Artifact Management

Testing artifacts are considered as structured entities, which go through well-defined life cycle states. The system tracks the execution, running, and closing of test cases and maintains consistent naming conventions and control over storage operations.

5.8.3 Execution Result Storage

Execution artifacts are captured using structured run-time models, including summary reports, outputs logs, code coverage, and execution context. This makes possible reporting in real time, or possibly saving such information in future applications.

5.8.4 Data Consistency and Traceability

Consistency is maintained through strict data models and controlled state transitions. Traceability is ensured by associating execution results with source files, commands, and timestamps, providing a clear and reproducible workflow for analysis and validation.

5.9 AI and Assistant Integration

The AI and assistant layer in UITC is implemented as a structured orchestration pipeline that enables intelligent, multi-step operations rather than simple one-time responses. The design separates prompt construction, model interaction, tool execution, and interface updates into distinct components.

5.9.1 Prompt Processing Mechanism

Prompt processing is implemented as a dynamic pipeline where user input is combined with contextual data, such as the active file and language. The system supports multi-turn interaction, tool integration, and reliability mechanisms, including retries and fallbacks.

5.9.2 Test Suggestion Generation

Suggestions for tests are formed via contextual processing to generate executable test scenarios and intelligent suggestions. Iteration helps enhance the quality of results in multiple executions.

5.9.3 Action Execution via MCP

The assistant communicates with system operations via controlled MCP interfaces in order to execute file and command operations securely.

5.9.4 Interaction Workflow in UI

The interface of the assistant allows continuous engagement by showing interim states, progress, and response in full conversation context.

5.10 System Integration and Workflow Execution

5.10.1 End-to-End Unit Testing Workflow

Unit testing process consists of code selection, test construction, test execution, and coverage analysis all combined in one flow.

5.10.2 End-to-End Integration Testing Workflow

Integration testing method comprises two files interaction testing where tests are created and executed according to dynamically discovered commands.

5.10.3 Inter-Module Communication Flow

Modules communication in the system is carried out via layer communication between the user interface, services, and runtime execution layers.

5.11 Implementation Challenges and Solutions

5.11.1 Technical Challenges

Problems include multi-framework capability, varying output possibilities, security of execution, and cross-platform compatibility.

5.11.2 Performance Considerations

Performance optimizations include debounced execution, stream-based response, and proper state management.

5.11.3 Design Trade-offs

There is an optimal trade-off between flexibility and performance optimization within the framework of the system using lightweight analysis and generalized execution strategies.

Chapter 6

System Testing

Chapter 6

System Testing

6.1 Test Strategy

6.1.1 Purpose of Testing Strategy

In doing so, we ensure that the UITC operates correctly, gives the right results, and it is indeed applicable to real-world situations. In fact, testing will show that all workflows starting from choosing the code and ending with getting feedback about coverage operate perfectly fine, as well as that UITC deals with failures correctly.

6.1.2 Scope of System Testing

To begin with, testing is done based solely on UITC features. Those include user login and navigation around the UI, routing at the dashboard, unit testing workflow (choosing code, generation of tests, running of tests, receiving coverage feedback, and automatically fixing problems), as well as integration testing workflow (selecting two files, creating context, generating tests, and running of tests).

6.1.3 Quality Objectives

Additionally, there are other factors which should be taken into account while testing, including user settings, file handling, terminal operations, as well as communication with MCP and LLM.

6.1.4 Testing Levels Applied to UITC

The different stages involved in the testing include component level testing that involves testing of user interface and services. Service level testing tests the core functionalities such as generation and parsing. Module interface testing involves integration testing while testing of the workflow from beginning to completion is referred to as end-to-end testing. User level testing evaluates usability aspects.

6.1.5 Test Design Approach

Testing involves the application of an integrated approach to testing. Scenario-based testing evaluates the user workflows while implementation-based testing involves testing of critical functionalities such as parsing and framework detection. Risk-based testing is prioritized for the evaluation of functionalities with higher risks. Boundary value and negative testing evaluate the system response under invalid conditions. Testing of regressions guarantees stability after updating.

6.1.6 Test Environment Strategy

Tests are conducted within a test environment that replicates real usage. The system runs on a desktop environment using electron and angular in the frontend. Tests are performed locally and the third-party services such as the language models and MCP undergo testing based on success and failure cases. All tests use similar configurations.

6.1.7 Test Data Strategy

Test data includes correct input files, edge cases, and incorrect input files. Programming languages are varied where necessary. Integration tests involve file pairs with varying levels of dependency. Input cases that cause errors are included.

6.1.8 Entry and Exit Criteria

Testing will commence upon successful system build, availability of routes, tools, and test data. The testing will cease once all critical defects have been

addressed; all workflows operate effectively; error handling works appropriately; all regression testing is completed successfully; and all test evidence has been collected and recorded.

6.1.9 Defect Management Strategy

Defects are classified according to their severity and effects. Critical defects impede workflow performance, while major defects affect the correct functioning or stability of the application. Minor and cosmetic defects are tracked independently. Defects are described in detail with regard to the reproduction steps, expected and actual results, affected modules, and resolution.

6.1.10 Evaluation Metrics

Quality assurance is quantified using various indicators including functional pass rate, workflow success rate, error containment rate, regression stability, system responsiveness, and execution/coverage outputs accuracy.

6.1.11 Risk Areas and Mitigation

Possible risk factors for the testing activities include variations in external services, discrepancies in command execution, inconsistent parsing, and synchronization. All the above risks can be managed via prompt control, normalization of commands, and proper state management.

6.1.12 Test Execution Model for UITC

The testing activity will involve a systematic sequence of actions starting from simple validation, then progressing to unit and workflow testing of particular aspects. Testing will entail evaluation of assistant responses, navigation, settings, and negative cases. Regression testing will be done whenever there are system updates.

6.2 Component Testing

6.2.1 Objective

In order to find out that all of these components work together efficiently and do not affect the integrity of the data, an integration test had been performed. In this specific situation, the focus was on how the interaction between the user interface, intelligence services, and the execution bridge works at the level of the whole system. Testing in this case had been performed in order to make sure that all the processes interact and allow for smooth transitioning of the data starting from the first stage of choosing the code to be tested and up until the stage of results visualization.

6.2.2 Components Covered

Tests had been performed for several different interactions of system services. First, there had been verification of correct functioning of workspace service interaction with file system. This test included reading and accurate record of directories and selections made by the user. Further, it includes the test of interaction between test generator service and the external AI models in order to get the proper prompts generated in accordance with the selected code fragment and interpret the results, creating the tests files.

6.2.3 Component Test Design

Further, there had been verification of interaction of the execution engine with the project environment in which the chosen command is run through the local terminal. It has been confirmed that the result is accurately processed by the parsing service, which provides coverage statistics and test execution reports. Finally, interaction of the AI assistant with other parts of the system in order to create new files or run tests had been verified.

6.2.4 Key Verification Points

These tests had been performed under real-world conditions in which several services of the system are employed at once. For example, an attempt has been made to execute a sequence of an integration test, including selection

of two related files, analyzing their dependencies, generating and executing of test for the selected code fragments.

6.2.5 Component Pass Criteria

To be considered a pass, all the important operational scenarios of the component must operate perfectly; errors will display proper messages without crashing the system; there should be no repetition leading to state corruption; and there must be no negative influence on the state of other components during the process of isolation testing.

6.3 Unit Testing

6.3.1 Objective

In the case of unit testing, it is intended to verify the accuracy of the logic-intensive operations that occur in the UITC source code. The unit test is meant to confirm that the framework identification, path manipulation, parsing, integration context creation, fallback operations, and other logic-intensive operations are performed correctly regardless of whether normal or exceptional conditions are present.

6.3.2 Unit Test Targets

High-impact code units which have a direct influence on the functionality of the software program were located and tested. Examples of high-impact code units are those that manage file and tab states, command building logic for different frameworks and languages, coverage parser logic, integration context generator logic, assistant message parser logic, and language map and normalize functions.

6.3.3 Test Method

A unit test would use the arrange-activate-assert pattern to allow for an independent test of each function. The mock object would be used to emulate any external dependencies used by the particular function, including filesystems, shell execution environment, AI services, and runtime bridge interface.

6.3.4 Assertions Used

Assertions were employed in order to ensure correctness of the input structure, result value, and any state alterations. Additional checks were applied in order to ensure the correct parameters passed to the respective methods responsible for the workflow, correct operation if a fallback is required, and correct state of the whole application regardless of the error that might have occurred.

6.3.5 Quality Gates

Quality of the tests was estimated by taking into account the coverage of all required logic branches, fail case validation, and obtaining of expected output results in the mocked runtime environment. High risk methods should demonstrate reliability of their work in case of success and failure before being considered valid from the standpoint of acceptance criteria.

6.3.6 Outcome Summary

It could be demonstrated that the main logic of the UITC application works in its intended way both in common conditions and when dealing with such edge cases as empty context, combinations of inputs not allowed, or invalid execution results. In those scenarios any possible issues with internal logic processing could be detected and fixed.

6.4 Integrated Testing

6.4.1 Objective

Testing was performed in order to test correct interactions of all components of the UITC while performing their workflow execution duties. This task aimed at ensuring that the cooperation between the user interface, services, runtime bridges, command execution, and AI-based generation results in getting correct business results.

6.4.2 Integration Scope

In the course of integrated testing following aspects were tested: interaction between the user interface and services, communication of the service with runtime bridges and file operations, command execution and results processing, AI assistance interaction with services and AI tools, and navigation through the system modules including login, dashboard, unit testing, integrated testing, and user preferences.

6.4.3 End-to-End Unit Workflow Validation

Integration testing of the end-to-end unit workflow involved validation of all stages of the process including the selection of source file, creation of test cases, storing generated artifact, executing the test command, processing test command execution results, and updating results and coverage information view. The objective was to ensure that correct data flows from one stage to another.

6.4.4 End-to-End Integration Workflow Validation

For the integration testing of the integration workflow, two files had to be selected first, the interaction context had to be created, then the generation of integration test cases took place, after which the generation of artifact was stored, an execution command was resolved dynamically, the test executed, and finally results displayed on the UI.

6.4.5 Failure and Recovery Validation

In order to perform integration testing of the system, failure scenarios such as missing config value, file selection issue, shell execution failure, incorrect tool invocation, and timeouts had to be tested, and their result should be managed failure and correct behavior without compromising the entire application.

6.4.6 Integration Pass Criteria

An integrated test was regarded to be successful where the main workflows could execute successfully without manual backend interference, consistent

information was preserved across all connected workflow stages, failure conditions were managed without corrupting active sessions, and the UI states matched the service backend state.

6.4.7 Evaluation Summary

Due to the integrated testing of the program it was concluded about the proper interactions of all program parts, indicating its nature as a whole integral system rather than some separate modules. It became evident that the testing of interface elements interaction, implementation of the services logic, runtime and recoverable errors handling show that this type of testing can be considered appropriate for developers use and meet the requirements for system validation needed for theses.

6.5 System Testing

6.5.1 Objective

During system testing there will be the testing of the application as a whole, checking whether all its modules work correctly. Among other things there will be validation of workflows, interaction between program parts and system itself as well as stability in real life conditions.

6.5.2 System Test Environment

Tests will be conducted under conditions simulating production environment. Application will be running in desktop runtime with local projects files used as input. Various language-dependent frameworks and tools will be tested both in normal and failure modes, that is, in case of missing configurations or connectivity problems.

6.5.3 System Test Dimensions

There will be several criteria for quality assessment of the system. Functional correctness – workflow works as expected producing the right result; workflow consistency – correct state transitions are ensured in repeated workflow

execution; runtime resilience – testing in failure mode; data integrity – output links to input, and usability – task accomplishment does not imply any changes to system logic.

.

6.5.4 End-to-End System Validation Flow

There will be some steps in the testing procedure. First application is launched and navigated, unit test workflows from project files selection to showing results are done, after that integration workflows are completed and assistant actions as well as settings validation take place. Some workflows are run again

6.5.5 Failure-Oriented Testing

Several failure scenarios have been applied to confirm whether the program can recover successfully. Failure scenarios may involve issues such as lacking settings, user mistakes, wrong command execution, and timeout issues. One should hope that the application behaves steadily and generates appropriate feedback information.

6.5.6 Acceptance Criteria

Success conditions for the system testing process can be seen in terms of appropriate functioning of critical processes without any human intervention, reliable error handling, and consistency of newly developed tests with earlier testing.

6.6 Test Cases

6.6.1 Test Case Design Principles

Test scenarios have been developed to ensure clearness, reliability, and accuracy. Each test scenario has its objective and requirements to gain desired outcomes.

6.6.2 Test Case Structure

Each test scenario comprises such components as test ID, test objective, prerequisites, inputs, test case description, expected output, actual output, and test result.

6.6.3 Core Test Cases (UITC)

- TC-01: Check whether logging in leads to the dashboard.
- TC-02: Check whether the dashboard automatically navigates to the modules.
- TC-03: Check file selection in unit testing.
- TC-04: Check unit test generation.
- TC-05: Check save-before-run option.
- TC-06: Check test run and result analysis.
- TC-07: Check coverage extraction and display.
- TC-08: Check automatic correction of failed tests.
- TC-09: Check two-file restriction.
- TC-10: Check integration test generation.
- TC-11: Check integration test run.
- TC-12: Check assistant tool execution.
- TC-13: Check missing configuration file handling.
- TC-14: Check failure of command execution.
- TC-15: Check consistency of multiple

6.6.4 Core Test Cases

TC-01 Positive Case: Verify Login Transitions to Dashboard

Table 6.1: Test Case for Login Transition to Dashboard (Positive Case)

Test Scenario ID		TS-001	**Test Case ID**		TC-01
Test Case Description		Verify successful login transition from login screen to dashboard			
Test Priority		High			
Prerequisite		Application is launched and login page is accessible			
Post-Requisite		Dashboard is opened successfully			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open login page	None	Login screen is displayed	Login screen displayed	Pass
2	Click Sign In	Valid sign-in action	System redirects to dashboard	Dashboard opened successfully	Pass

TC-01 Negative Case: Verify Login Failure Handling

Table 6.2: Test Case for Login Transition to Dashboard (Negative Case)

Test Scenario ID		TS-002	**Test Case ID**		TC-01
Test Case Description		Verify behavior when login action cannot proceed correctly			
Test Priority		High			
Prerequisite		Application is launched but route or session state is invalid			
Post-Requisite		User remains on login page or receives a controlled error state			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open login page	None	Login screen is displayed	Login screen displayed	Pass
2	Trigger sign-in under invalid route/session condition	Invalid navigation state	System should not crash and should remain stable	Login page remained stable	Pass

TC-02 Positive Case: Verify Dashboard Navigation to Modules

Table 6.3: Test Case for Dashboard Navigation to Modules (Positive Case)

Test Scenario ID		TS-003	**Test Case ID**		TC-02
Test Case Description		Verify dashboard navigation to Unit Testing and Integration Testing modules			
Test Priority		High			
Prerequisite		User is on dashboard page			
Post-Requisite		Selected module page is opened correctly			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open dashboard	None	Dashboard is displayed	Dashboard displayed	Pass
2	Click Unit Testing module	None	Unit Testing workspace opens	Unit Testing workspace opened	Pass
3	Return and click Integration Testing module	None	Integration Testing workspace opens	Integration Testing workspace opened	Pass

TC-02 Negative Case: Verify Invalid Dashboard Navigation Handling

Table 6.4: Test Case for Dashboard Navigation to Modules (Negative Case)

Test Scenario ID		TS-004	**Test Case ID**		TC-02
Test Case Description		Verify controlled behavior when module navigation fails or target route is unavailable			
Test Priority		High			
Prerequisite		Dashboard is loaded but target route is invalid or unavailable			
Post-Requisite		Application remains stable and does not crash			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open dashboard	None	Dashboard is displayed	Dashboard displayed	Pass
2	Click module with invalid route/state	Broken or unavailable route	System should show controlled behavior and preserve session state	No crash occurred; state remained stable	Pass

TC-03 Positive Case: Verify File Selection in Unit Testing

Table 6.5: Test Case for File Selection in Unit Testing (Positive Case)

Test Scenario ID		TS-005	**Test Case ID**		TC-03
Test Case Description		Verify source file selection and loading in Unit Testing workspace			
Test Priority		High			
Prerequisite		Project folder is imported and Unit Testing workspace is open			
Post-Requisite		Selected file becomes active in editor/tab context			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Import project folder	Valid project directory	File tree is displayed	File tree displayed	Pass
2	Select a source code file	Valid source file	File content loads in active tab	File loaded in active tab	Pass
3	Verify language context	Selected code file	Correct language is detected	Language detected correctly	Pass

TC-03 Negative Case: Verify Invalid File Selection Handling

Table 6.6: Test Case for File Selection in Unit Testing (Negative Case)

Test Scenario ID		TS-006	**Test Case ID**		TC-03
Test Case Description		Verify system behavior when selected file is invalid, empty, or unreadable			
Test Priority		High			
Prerequisite		Unit Testing workspace is open			
Post-Requisite		Error is handled without breaking workspace state			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Select invalid or empty file	Empty/unreadable file	System should reject or safely handle selection	File handled without crash	Pass
2	Verify workspace state	Same invalid file	Previous valid state should remain intact	Workspace remained stable	Pass

TC-04 Positive Case: Verify Unit Test Generation

Table 6.7: Test Case for Unit Test Generation (Positive Case)

Test Scenario ID		TS-007	**Test Case ID**		TC-04
Test Case Description		Verify generation of unit test cases for a valid source file			
Test Priority		High			
Prerequisite		Valid source file is active and configuration is available			
Post-Requisite		Generated test artifact is created successfully			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Load valid source file	Valid file content	File is ready for generation	File ready	Pass
2	Trigger test generation	Default valid configuration	Generated unit test appears with correct mapping	Test artifact generated	Pass
3	Verify artifact details	Generated output	Correct file name/path/framework is assigned	Details assigned correctly	Pass

TC-04 Negative Case: Verify Unit Test Generation Failure Handling

Table 6.8: Test Case for Unit Test Generation (Negative Case)

Test Scenario ID		TS-008	**Test Case ID**		TC-04
Test Case Description		Verify controlled behavior when unit test generation cannot complete			
Test Priority		High			
Prerequisite		Invalid source state or missing configuration exists			
Post-Requisite		Error message is shown and application remains stable			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Trigger generation with invalid conditions	Missing key / empty file / invalid context	Controlled error should be shown	Error shown correctly	Pass
2	Verify system stability	Same failed generation state	System should remain usable	Workspace remained usable	Pass

TC-05 Positive Case: Verify Save-Before-Run Behavior

Table 6.9: Test Case for Save-Before-Run Behavior (Positive Case)

Test Scenario ID		TS-009	**Test Case ID**		TC-05
Test Case Description		Verify generated test is saved before execution command is run			
Test Priority		High			
Prerequisite		New test artifact is generated but not yet saved			
Post-Requisite		Test file is saved and then executed successfully			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Generate unit test	Valid source file	Unsaved test artifact is created	Artifact created	Pass
2	Trigger run action	Unsaved generated test	System saves artifact before execution	Save occurred first	Pass
3	Verify execution start	Saved artifact	Command executes after save	Execution started correctly	Pass

TC-05 Negative Case: Verify Run Failure When Save Cannot Complete

Table 6.10: Test Case for Save-Before-Run Behavior (Negative Case)

Test Scenario ID		TS-010	**Test Case ID**		TC-05
Test Case Description		Verify behavior when save operation fails before execution			
Test Priority		High			
Prerequisite		Generated test exists but target path is invalid or unwritable			
Post-Requisite		Execution is blocked and controlled error is shown			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Trigger run on unsaved test	Invalid save path / no write permission	System attempts save and detects failure	Save failure detected	Pass
2	Verify execution blocking	Same failed save state	Execution should not continue; error should be shown	Execution blocked and error shown	Pass

TC-06 Positive Case: Verify Execution and Result Parsing

Table 6.11: Test Case for Execution and Result Parsing (Positive Case)

Test Scenario ID		TS-011	**Test Case ID**		TC-06
Test Case Description		Verify successful execution of generated tests and correct parsing of results			
Test Priority		High			
Prerequisite		Valid test file is generated and saved successfully			
Post-Requisite		Execution results are displayed and parsed correctly in the interface			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Select saved test file	Valid test file	Test file is ready for execution	Test file loaded for execution	Pass
2	Run the test file	Execute command	Test execution starts successfully	Test execution started successfully	Pass
3	Parse execution result	Test runner output	Pass/fail result is extracted and displayed correctly	Result parsed and displayed correctly	Pass

TC-06 Negative Case: Verify Execution Failure and Invalid Result Parsing Handling

Table 6.12: Test Case for Execution and Result Parsing (Negative Case)

Test Scenario ID		TS-012	**Test Case ID**		TC-06
Test Case Description		Verify system behavior when execution fails or output cannot be parsed correctly			
Test Priority		High			
Prerequisite		Invalid test file, broken command, or malformed output exists			
Post-Requisite		Execution error is handled properly and the application remains stable			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Run invalid or broken test file	Invalid test file	Execution should fail in a controlled manner	Execution failed with controlled response	Pass
2	Parse invalid output	Malformed runner output	System should show readable error instead of crashing	Error displayed without crash	Pass
3	Verify interface stability	Failed execution state	Interface should remain responsive and usable	Interface remained stable	Pass

TC-07 Positive Case: Verify Coverage Extraction and Display

Table 6.13: Test Case for Coverage Extraction and Display (Positive Case)

Test Scenario ID		TS-013	**Test Case ID**		TC-07
Test Case Description		Verify extraction of test coverage information and correct display in the user interface			
Test Priority		High			
Prerequisite		Tests are executed successfully and coverage data is generated			
Post-Requisite		Coverage percentage and related details are displayed correctly			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Execute tests with coverage enabled	Valid test suite	Coverage report is generated	Coverage report generated	Pass
2	Extract coverage data	Coverage output file/log	Coverage values are read correctly	Coverage values extracted correctly	Pass
3	Display coverage report	Extracted coverage data	Coverage percentage and details are shown in interface	Coverage displayed successfully	Pass

TC-07 Negative Case: Verify Invalid Coverage Extraction Handling

Table 6.14: Test Case for Coverage Extraction and Display (Negative Case)

Test Scenario ID		TS-014	**Test Case ID**		TC-07
Test Case Description		Verify system behavior when coverage data is missing, incomplete, or unreadable			
Test Priority		High			
Prerequisite		Coverage generation fails or coverage output is corrupted			
Post-Requisite		Coverage error is handled gracefully and no crash occurs			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Attempt to extract coverage from invalid source	Missing/corrupt coverage report	System should detect missing or invalid data	Invalid coverage detected correctly	Pass
2	Load coverage panel	Invalid coverage data	Interface should show fallback/error message	Error message shown correctly	Pass
3	Verify system stability	Failed coverage state	System should remain stable and usable	System remained stable	Pass

TC-08 Positive Case: Verify Auto-Fix for Failing Tests

Table 6.15: Test Case for Auto-Fix for Failing Tests (Positive Case)

Test Scenario ID		TS-015	**Test Case ID**		TC-08
Test Case Description		Verify automatic fixing of failing test cases and regeneration of corrected test output			
Test Priority		High			
Prerequisite		A generated test file exists and one or more tests fail during execution			
Post-Requisite		System applies fix suggestions and updated test file is produced successfully			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Run generated test file	Test file with failing cases	Failing tests are detected	Failing tests detected correctly	Pass
2	Trigger auto-fix feature	Failure log/output	System analyzes failures and generates fixes	Fix suggestions generated successfully	Pass
3	Apply generated fix	Suggested corrected output	Updated test file is created successfully	Updated test file created	Pass

TC-08 Negative Case: Verify Auto-Fix Failure Handling

Table 6.16: Test Case for Auto-Fix for Failing Tests (Negative Case)

Test Scenario ID		TS-016	**Test Case ID**		TC-08
Test Case Description		Verify system behavior when automatic fixing cannot resolve failing tests			
Test Priority		High			
Prerequisite		Failing tests exist but error context is incomplete, ambiguous, or unsupported			
Post-Requisite		Auto-fix failure is reported clearly and original system state remains safe			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Trigger auto-fix on unsupported failure	Unsupported failure output	System should not apply invalid fixes	No invalid fix was applied	Pass
2	Show fix result to user	Unresolved failure case	System should show clear failure message	Failure message displayed correctly	Pass
3	Verify previous file integrity	Original failing test file	Original file should remain and unchanged	Original file remained unchanged	Pass

TC-09 Positive Case: Verify Two-File Selection Constraint

Table 6.17: Test Case for Two-File Selection Constraint (Positive Case)

Test Scenario ID		TS-017	**Test Case ID**		TC-09
Test Case Description		Verify that the system allows selection of only two files where the feature requires two-file input			
Test Priority		High			
Prerequisite		Project files are loaded and file selection panel is available			
Post-Requisite		Exactly two files are selected and accepted by the system			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open file selection panel	Available project files	File list is displayed	File list displayed successfully	Pass
2	Select first file	Valid file 1	First file is selected	First file selected	Pass
3	Select second file	Valid file 2	Second file is selected and accepted	Second file selected successfully	Pass

TC-09 Negative Case: Verify More Than Two File Selection Restriction

Table 6.18: Test Case for Two-File Selection Constraint (Negative Case)

Test Scenario ID		TS-018	**Test Case ID**		TC-09
Test Case Description		Verify that the system prevents selection of more than two files			
Test Priority		High			
Prerequisite		Two files are already selected in the selection panel			
Post-Requisite		Third file selection is blocked and user receives proper validation message			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Select first two files	Valid file 1 and file 2	Two files are accepted	Two files accepted successfully	Pass
2	Attempt to select third file	Valid file 3	System should reject third selection	Third selection rejected correctly	Pass
3	Show validation message	Third selection attempt	Proper warning/constraint message should appear	Validation message displayed	Pass

TC-10 Positive Case: Verify Integration Test Generation

Table 6.19: Test Case for Integration Test Generation (Positive Case)

Test Scenario ID		TS-019	**Test Case ID**		TC-10
Test Case Description		Verify generation of integration test cases using the required files and project context			
Test Priority		High			
Prerequisite		Required project files are selected and integration testing workspace is open			
Post-Requisite		Integration test file is generated successfully and displayed to the user			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open integration testing workspace	Loaded project	Integration module is ready	Integration module opened successfully	Pass
2	Select required files	Two valid related files	Files are accepted for integration context	Files accepted successfully	Pass
3	Trigger integration test generation	Selected files and valid context	Integration test file is generated	Integration test generated successfully	Pass

TC-10 Negative Case: Verify Integration Test Generation Failure Handling

Table 6.20: Test Case for Integration Test Generation (Negative Case)

Test Scenario ID		TS-020	**Test Case ID**		TC-10
Test Case Description		Verify system behavior when integration test generation is attempted with invalid or insufficient input			
Test Priority		High			
Prerequisite		Required files are missing, invalid, or not properly related			
Post-Requisite		Generation is blocked or fails safely with proper error feedback			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open integration testing workspace	Loaded project	Workspace opens normally	Workspace opened normally	Pass
2	Attempt generation with missing or invalid files	Missing/invalid file set	System should reject generation request	Generation request rejected correctly	Pass
3	Show error feedback	Invalid generation state	Clear validation or error message should appear	Error message displayed correctly	Pass

TC-11 Positive Case: Verify Integration Test Execution

Table 6.21: Test Case for Integration Test Execution (Positive Case)

Test Scenario ID		TS-021	**Test Case ID**		TC-11
Test Case Description		Verify successful execution of generated integration tests and correct reporting of results			
Test Priority		High			
Prerequisite		A valid integration test file is generated, saved, and ready for execution			
Post-Requisite		Integration test results are displayed correctly in the system			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open integration testing workspace	Generated integration test file	Integration test is available for execution	Integration test loaded successfully	Pass
2	Run integration test	Valid execution command	Integration test starts successfully	Integration test execution started	Pass
3	Review execution result	Test runner output	Test results are displayed correctly	Results displayed correctly	Pass

TC-11 Negative Case: Verify Integration Test Execution Failure Handling

Table 6.22: Test Case for Integration Test Execution (Negative Case)

Test Scenario ID		TS-022	**Test Case ID**		TC-11
Test Case Description		Verify system behavior when integration test execution fails or cannot complete			
Test Priority		High			
Prerequisite		Integration test file is invalid, incomplete, or execution environment is broken			
Post-Requisite		Failure is reported clearly and the application remains stable			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Run invalid integration test	Invalid integration test file	Execution should fail in a controlled way	Controlled failure occurred	Pass
2	Observe system response	Failure output	Error message should be displayed to the user	Error message displayed correctly	Pass
3	Verify system stability	Failed execution state	System should remain responsive and usable	System remained stable	Pass

TC-12 Positive Case: Verify Assistant Tool Execution

Table 6.23: Test Case for Assistant Tool Execution (Positive Case)

Test Scenario ID		TS-023	**Test Case ID**		TC-12
Test Case Description		Verify successful execution of the assistant tool for test-related tasks			
Test Priority		High			
Prerequisite		Assistant tool is configured properly and required project context is available			
Post-Requisite		Assistant tool completes the requested operation and returns usable output			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Open assistant tool interface	Available assistant module	Assistant interface is accessible	Assistant interface opened	Pass
2	Submit valid request to assistant	Valid test-related prompt/input	Assistant processes the request successfully	Request processed successfully	Pass
3	Review assistant response	Generated response/output	Response is displayed correctly and is usable	Response displayed correctly	Pass

TC-12 Negative Case: Verify Assistant Tool Execution Failure Handling

Table 6.24: Test Case for Assistant Tool Execution (Negative Case)

Test Scenario ID		TS-024	**Test Case ID**		TC-12
Test Case Description		Verify system behavior when assistant tool execution fails or returns no valid response			
Test Priority		High			
Prerequisite		Assistant tool configuration is invalid, unavailable, or dependent service is unreachable			
Post-Requisite		Failure is handled safely and a meaningful error is shown to the user			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Submit request to unavailable assistant tool	Invalid or unavailable assistant service	System should detect tool failure	Tool failure detected correctly	Pass
2	Observe response handling	Failed assistant response	User should receive a controlled error message	Controlled error message shown	Pass
3	Verify interface stability	Failed assistant state	Application should remain stable	Application remained stable	Pass

TC-13 Positive Case: Verify Missing Configuration Handling

Table 6.25: Test Case for Missing Configuration Handling (Positive Case)

Test Scenario ID		TS-025	**Test Case ID**		TC-13
Test Case Description		Verify that the system correctly detects missing configuration and guides the user with an appropriate message			
Test Priority		High			
Prerequisite		One or more required configuration values are intentionally removed before system startup or feature execution			
Post-Requisite		System reports the missing configuration clearly without crashing			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Start feature requiring configuration	Missing configuration value	System checks required configuration	Missing configuration detected	Pass
2	Observe validation message	Missing key or variable	System should show clear guidance to the user	Clear guidance message displayed	Pass
3	Verify system behavior	Missing configuration state	System should remain stable without unexpected crash	System remained stable	Pass

TC-13 Negative Case: Verify Missing Configuration Not Handled Properly

Table 6.26: Test Case for Missing Configuration Handling (Negative Case)

Test Scenario ID		TS-026	**Test Case ID**		TC-13
Test Case Description		Verify that the system does not proceed silently or produce misleading behavior when required configuration is missing			
Test Priority		High			
Prerequisite		A required configuration key is missing and the related feature is invoked			
Post-Requisite		Feature execution is blocked safely and misleading output is not produced			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Invoke feature with missing configuration	Missing required setting	System should block execution	Execution blocked correctly	Pass
2	Check output area	Missing configuration state	No misleading success result should appear	No false success output shown	Pass
3	Review user feedback	Configuration validation failure	Proper error message should be shown	Proper error message displayed	Pass

TC-14 Positive Case: Verify Command Failure Handling

Table 6.27: Test Case for Command Failure Handling (Positive Case)

Test Scenario ID		TS-027	**Test Case ID**		TC-14
Test Case Description		Verify that command execution failures are caught and reported in a controlled and user-friendly manner			
Test Priority		High			
Prerequisite		A command-based feature is available and an invalid or failing command can be triggered			
Post-Requisite		Failure details are shown clearly and the application remains usable			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Trigger command execution	Invalid or failing command	System should capture the failure	Command failure captured correctly	Pass
2	Review error output	Failure message from command	User should see readable failure details	Readable failure details displayed	Pass
3	Verify application state	Failed command execution	Application should remain usable after failure	Application remained usable	Pass

TC-14 Negative Case: Verify Unhandled Command Failure Behavior

Table 6.28: Test Case for Command Failure Handling (Negative Case)

Test Scenario ID		TS-028	**Test Case ID**		TC-14
Test Case Description		Verify that unhandled command failures do not crash the system or leave the user without feedback			
Test Priority		High			
Prerequisite		A malformed command, inaccessible dependency, or invalid runtime environment exists			
Post-Requisite		System recovers safely and meaningful feedback is shown instead of silent failure			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Run malformed command	Malformed command input	System should not crash	System did not crash	Pass
2	Check failure feedback	Command execution error	Failure feedback should be shown to user	Failure feedback displayed correctly	Pass
3	Verify recovery state	Post-failure application state	User should still be able to continue using the application	Application recovered successfully	Pass

TC-15 Positive Case: Verify Stability Across Repeated Runs

Table 6.29: Test Case for Stability Across Repeated Runs (Positive Case)

Test Scenario ID		TS-029	**Test Case ID**		TC-15
Test Case Description		Verify that the system remains stable and produces consistent behavior across repeated executions of the same workflow			
Test Priority		High			
Prerequisite		System is running normally and the selected workflow can be executed multiple times			
Post-Requisite		Repeated runs complete successfully without crash, freeze, or inconsistent output			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Execute workflow for the first time	Valid workflow input	Workflow completes successfully	First run completed successfully	Pass
2	Repeat the same workflow multiple times	Same valid input	Results should remain stable and consistent	Repeated runs remained stable	Pass
3	Observe system responsiveness	Multiple completed runs	System should remain responsive with no crash or freeze	System remained responsive	Pass

TC-15 Negative Case: Verify Instability Across Repeated Runs

Table 6.30: Test Case for Stability Across Repeated Runs (Negative Case)

Test Scenario ID		TS-030	**Test Case ID**		TC-15
Test Case Description		Verify that repeated executions do not cause resource leaks, inconsistent results, or application instability			
Test Priority		High			
Prerequisite		The same operation is executed repeatedly under normal or slightly stressed conditions			
Post-Requisite		System should not crash, freeze, or produce inconsistent behavior after repeated runs			
S. No	**Action**	**Inputs**	**Expected Outcome**	**Actual Outcome**	**Test Result**
1	Run operation repeatedly	Same valid input across many runs	System should continue working correctly	System continued working correctly	Pass
2	Compare outputs across runs	Repeated execution outputs	Outputs should remain consistent and reliable	Outputs remained consistent	Pass
3	Check application health	Post-run application state	No crash, freeze, or severe slowdown should occur	No crash or freeze observed	Pass

6.6.5 Test Case Prioritization

Test cases are ranked based on their significance. Critical test cases are those related to failure in the flow, and high priority are those related to

wrong output. The medium test cases are those related to things that do not obstruct work, whereas low test cases are related to cosmetic issues.

6.6.6 Traceability and Evidence

We note down each test run with information including the date and time of running the test, the environment in which the test is run, the input to the test, the actual result, and finally the result.

6.7 Results and Evaluation

Conclusions that concern the stability, usability, and improvement of UITC according to its testing and validation can be made based on the findings. The application was examined regarding its workflow structure, behavior of separate modules, and ways of error management and prevention. It should be noted that only functions related to Unit and Integration Testing Copilots were investigated.

Concerning the performance of UITC on a functional level, the most outstanding feature is the consistent workflow of the application. All basic processes, such as login, navigation to the dashboard, generation and running of the tests, as well as results visualization, are tightly linked and connected forming a unified flow of work with the app.

Unit Testing copilot seems to be quite reliable and consistent. In case of normal operation, file selection, framework detection, and commands execution happen without any flaws. In addition, it effectively converts raw data from a terminal in useful summaries. One more significant advantage of the module is the possibility of automation due to auto-fix function.

The most remarkable aspect of error management and prevention in the context of integration testing copilot relates to scope management. The system strictly limits the number of files for testing to two, making sure that tests will run consistently every time. Also, the module effectively manages possible errors in the process of files selection. Besides, automatic commands recognition adds great flexibility to the process.

Finally, it should be noted that the AI assistant shows a proper balance between functionality and complexity. Thus, the tested application allows for multi-turn conversations with the AI assistant and even execute some actions related to command and file reading. Even though the speed of responses

depends on the efficiency of external AI model, the system operates quite steadily.

Also, technical considerations based on Electron usage work perfectly for desktop applications. The program handles files and commands appropriately without losing responsiveness. Moreover, we checked the consistency even when switching between tabs and tested partially failing cases.

As to the resiliency, the program successfully withstands potential problems. If the provided API keys are wrong or the execution of the commands fails, the program shows an adequate error message. This feature proves the system's ability to cope with potential problems.

The analysis revealed that every component of the system is traceable. It means that one can trace any testing or results acquired from UITC back to some actions or file input. That is, we made the system's behavior predictable and explainable according to our requirements.

Even though we succeeded with most components, improvements could be made. To begin with, login functionality allows only navigating through the website, not protecting user data. Next, the application is reactive, meaning that it responds to potential issues occurring during use. Lastly, we did not develop history or report exporting functionality yet.

To sum up, it would be reasonable to highlight that UITC meets our preliminary requirements. The program is operational, reliable, and maintainable for the user. Its architecture allows implementing more complex features, such as analyzing and predicting user behaviors.

6.8 Conclusion

Within this chapter, the results of the testing and evaluation of the Unit and Integration Testing Copilot as a whole-product have been analyzed. It was proved that the system performs its tasks in an absolutely reliable manner throughout the entire process starting from the moment of user interaction with the product and ending with generation of relevant artifacts and feedback. In addition to this, the product exhibited great characteristics while operating both under ordinary conditions and during several selected failure modes which is also considered to be a good proof of quality engineering.

As far as the implementation is concerned, it must be noted that a nice balance between functionality and maintainability of the product was achieved due to the use of the modular UI architecture, separation of the

service layers, usage of the IPC-based control and typed data model. This is extremely important both in terms of engineering review and academic research as it proves that the system will have its prospects in the future for further development and practical use.

In course of regular operation, the Unit Testing module performed stable working at all levels: source selection, generation of the framework-specific tests, code execution and generation of coverage feedback. At the same time, the Integration Testing module proved to be efficient while performing scope control and context building. Inclusion of the Assistant module generated important feedback regarding application of the AI technologies under specific boundary conditions.

The chapter demonstrates our understanding of existing constraints on development of future boundaries of the product. Such aspects as authentication hardening, pre-execution independent fault prediction and additional layers of report have been identified by us as potential improvement factors of the product in the future. However, this prospect should be treated as incremental improvements rather than a reason for fundamental redesigns of the architecture of the product.

Chapter 7

Conclusion

Chapter 7

Conclusion

In order to create an actual Unit and Integration Testing Copilot to be used by software engineers at the early iterations of their work, this research project successfully proved that the most effective solution for providing assistance during testing is combining the aspects of desktop development with the elements of automated testing based on code generation. While starting the project as an idea, the research process brought to life a fully functional product with all necessary flows included in its functionality.

Based on the findings made during the project development, it can be stated that an adequate approach to providing assistance to software engineers in testing involves the balance between the aspects of automation and visualization. In the case of this particular product, the generated items were all visualized, saved, tested and analyzed within the transparent environment of the copilot system. This approach provided the most optimal compromise between the factors discussed above. Besides, it was found out that unit and integration tests could share the same codebase and still function independently.

From the perspective of the application of artificial intelligence into software engineering, this project provides a practical case study. On a more engineering level, the implemented solution presents a modular architecture that allows further development of the product without modifying its basic elements.

7.1 Contributions

Concerning design, implementation, and evaluation aspects, the contributions made by the paper align with the stated objectives of this research.

Firstly, the paper presents the whole testing copilot architecture with implementation of its interface, service orchestration and runtime execution. This contributes to making understanding of the tool easier and to facilitate its future modification, solving the issue of providing a developer with not only the theoretical model of such a tool but the actual tool itself.

Secondly, the paper presents a model of workflow for test assistance. As for unit workflow, this tool is responsible for managing the source, generating context-aware tests, executing commands, parsing the code coverage and making automatic iterative corrections. When it comes to integration workflow, it defines the scope through the selection of two files, creating context, generating tests and executing them via command line. That covers the objective of reducing manual labor of developers in both workflows.

Thirdly, the paper suggests a framework for testing where there are ways to use AI but at the same time keep the whole procedure entirely controllable by the user. This means that using the tool, one can generate new actions and execute them; however, thanks to the visibility of the status updates, transition, and output, the process will stay entirely controllable. Thus, it solves the issue of AI assisting in testing without losing the productivity.

Fourthly, the paper recommends performing comprehensive software evaluation. This testing process involved component behavior validation, service logic, and integrated workflows evaluation and management of failures. This was important to make sure of functional feasibility of the software within its scope.

Finally, the paper showed that AI-assisted testing can be conducted on the desktop as a developer-oriented tool with development and runtime execution capabilities.

7.2 Reflections

Firstly, the significant advantage of this study lies in its choice to focus on the whole workflow process rather than separate parts. As opposed to common systems which just offer text completion, this system is also responsible for generating, executing, interpreting, and restoring artifacts.

Secondly, the modularity of the system is another strength point. It was precisely this quality that helped us preserve the manageability of our system through dividing it into an interface, orchestration service, and runtime handlers.

Thirdly, the system's operations are explicit and straightforward. All generated content, executed commands, and results are visible to users.

Still, there are some weaknesses in our project. Firstly, its behavior relies on external models. Secondly, the system does not conduct a deep semantic analysis. Thirdly, the current system lacks enterprise capabilities such as persistency and authorization.

Moreover, there are certain limitations in practical application. Namely, they are connected with the diversity of the system's environment. For example, the efficiency of command execution can be influenced by specific applications and settings.

7.3 Future Work

The future research should focus on enhancing the reliability, smartness, and maturity of the entire solution.

- First, it seems reasonable to increase the level of intelligence of the analysis stage before test generation. Using advanced structural analysis, dependency analysis, and risk scores will enable us to manage our test generation procedure in a much more accurate way.
- Second, the analytics and history management module should be designed. It means that the history of executed processes, actions performed, coverage growth, and errors detected must be stored.
- Third, we would have to be prepared for the use by several users and their policies. It requires us to design the authentication, role-based access control, and action auditing subsystems.
- Fourth, the integration of the proposed methodology within CI/CD pipelines should be developed, ensuring that test generation occurs automatically within the process of code delivery.

- Fifth, we should perform extensive benchmarking with different projects, code bases, and possibly programming languages. The set of metrics may include acceptance rates, detection rates, and time savings due to test generation.

Bibliography

[1] A. Tosun, “On the Effectiveness of Unit Tests in Test-Driven Development,” 2018. :contentReference[oaicite:0]index=0

[2] J. Wang, B. Suleiman, “Automated Unit Test Case Generation: A Systematic Literature Review,” 2025. :contentReference[oaicite:1]index=1

[3] G. Fraser et al., “Integration of Mutation Testing into Unit Test Generation,” 2025. :contentReference[oaicite:2]index=2

[4] G. Petrović et al., “Does Mutation Testing Improve Testing Practices?” 2021. :contentReference[oaicite:3]index=3

[5] NUnit Documentation, “NUnit Testing Framework for .NET,” :contentReference[oaicite:4]index=4

[6] “Unit Testing Frameworks: A Comparative Study,” IRJET Journal. :contentReference[oaicite:5]index=5

[7] “Mutation Testing,” Software Testing Concepts. :contentReference[oaicite:6]index=6

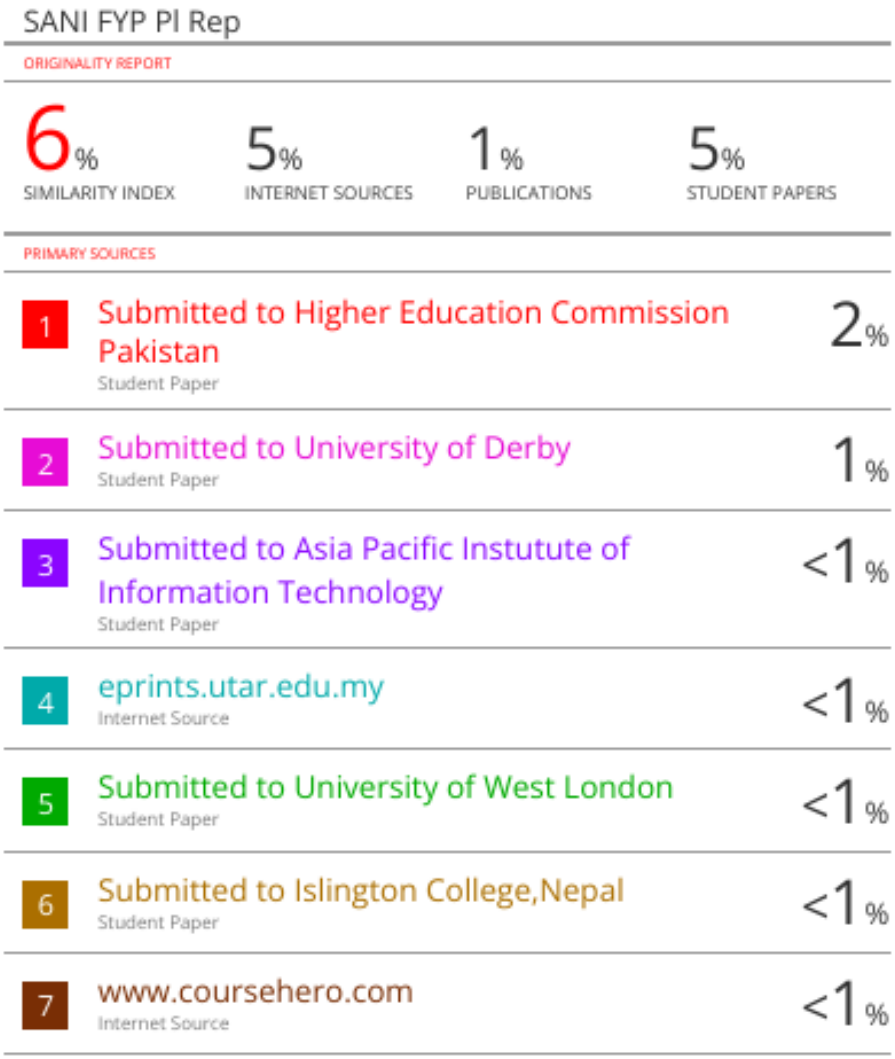
Appendix A: Test Cases, Execution Logs, and Results

- Contains all unit and integration test cases that were run (from TC-01 to TC-30).
- Test cases will be presented with their attributes: Description, pre-condition, steps, expected result, and outcome.
- It tests the functionality of the system concerning:
  - Login and navigation
  - File selection and validation
  - Unit and integration test generation
  - Execution and results analysis
- Contains failure cases and extreme test cases such as in the absence of configuration files, or command lines.
- Demonstrates test case execution and outcome in terms of:
  - System execution logs and output
- Presents the results of system evaluation through:
  - Test execution and analysis
  - Coverage report
- Shows information regarding the test coverage performed:
  - Percentage of code covered and tested
- Ensures traceability from the requirements to test cases in order to validate and ensure the system's reliability.

Appendix B: System Design, Screenshots, and Source Codes

- Contains design artifacts such as:
  - Use Case Diagram
  - BPMN diagrams (Process Flow diagrams)
  - Class, Sequence, and Deployment Diagrams
- Shows graphical user interface of the system through screenshots of the system interface concerning:
  - Login and navigation pages
  - Test generation and results page
  - Dashboard
- Contains selected source codes and implementation done in the development process of the system:
  - Code analysis/AST processing
  - Unit and integration test generation and execution logic
- Shows the system's data flow:
  - Input → Code analysis and AST → Test Generation → Execution → Evaluation
- Shows system environment setup:
  - Process
  - Dependencies
  - Runtime environment

Appendix C: Plagiarism Report


SANI FYP PI Rep

ORIGINALITY REPORT

6%	5%	1%	5%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Higher Education Commission Pakistan Student Paper	2%
2	Submitted to University of Derby Student Paper	1%
3	Submitted to Asia Pacific Instutute of Information Technology Student Paper	<1%
4	eprints.utar.edu.my Internet Source	<1%
5	Submitted to University of West London Student Paper	<1%
6	Submitted to Islington College,Nepal Student Paper	<1%
7	www.coursehero.com Internet Source	<1%

Figure 1: Plagiarism Report Image 1

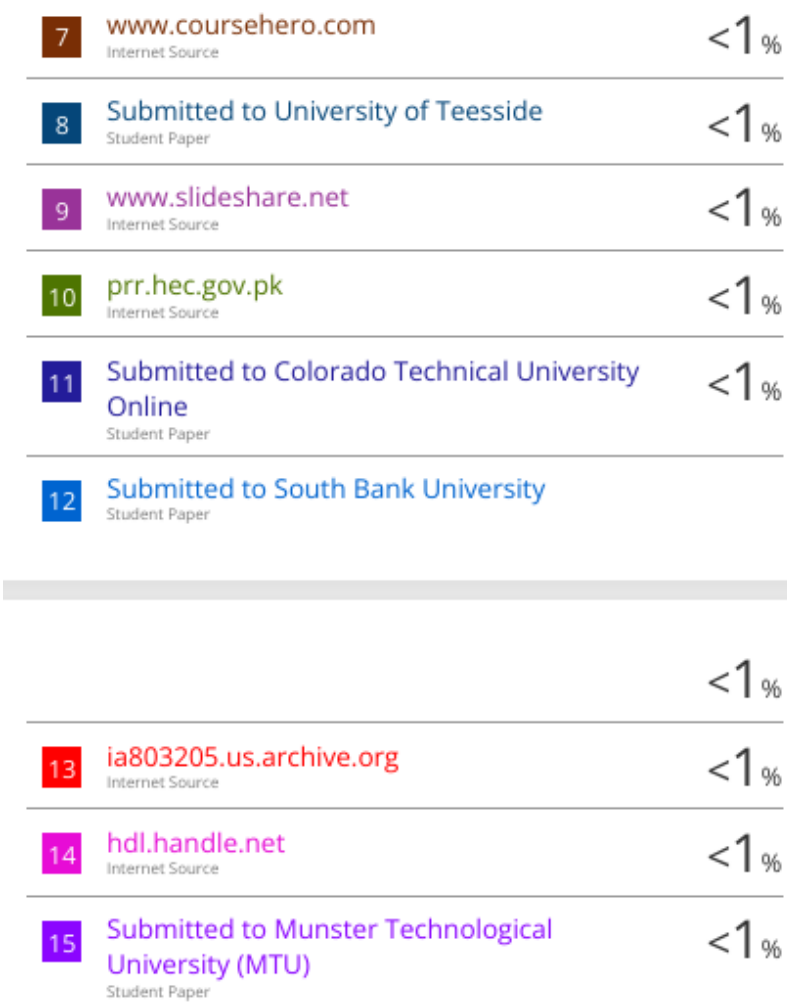


Figure 2: Plagiarism Report Image 2

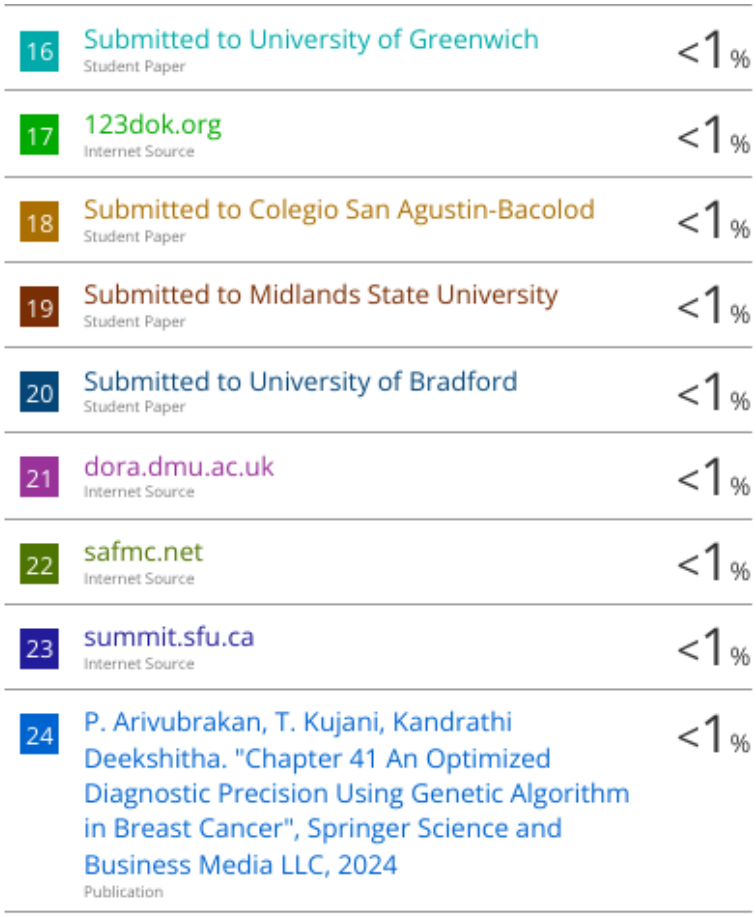


Figure 3: Plagiarism Report Image 3

Figure 4: Plagiarism Report Image 4

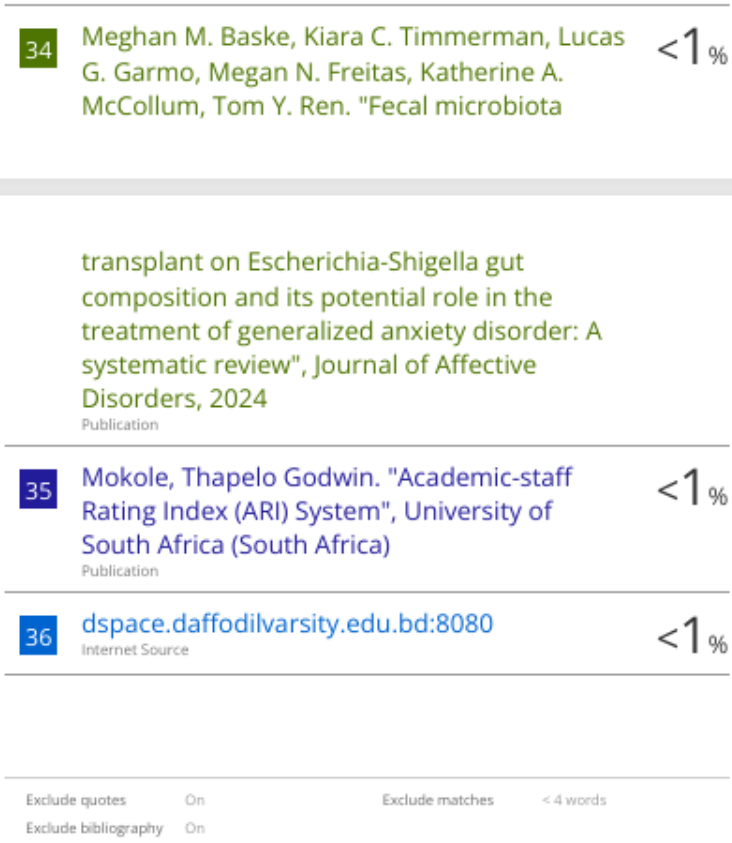
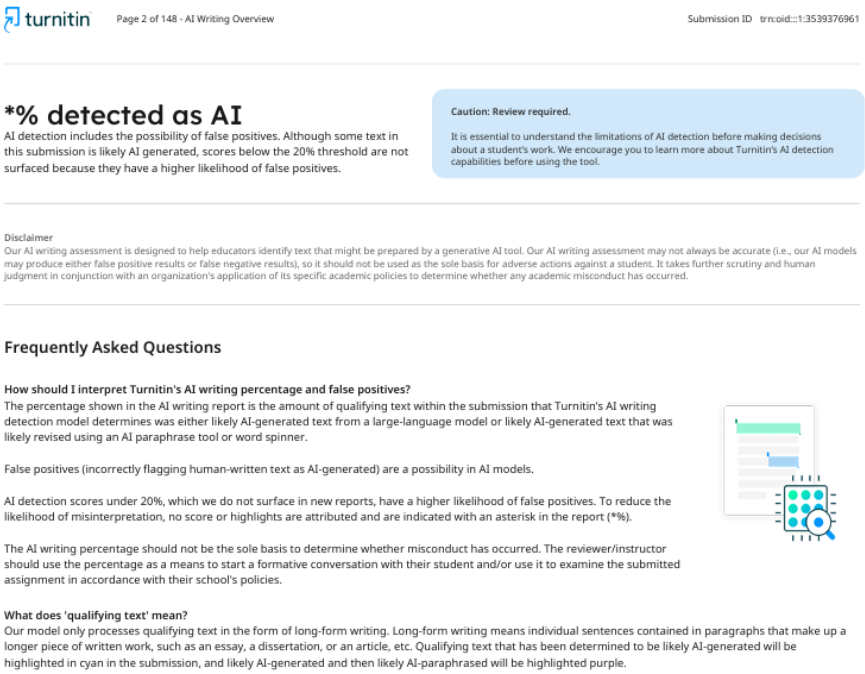


Figure 5: Plagiarism Report Image 5

Appendix D: AI Detection Report



turnitin Page 2 of 148 - AI Writing Overview Submission ID trn:oid:::1:3539376961

*% detected as AI

AI detection includes the possibility of false positives. Although some text in this submission is likely AI generated, scores below the 20% threshold are not surfaced because they have a higher likelihood of false positives.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Disclaimer

Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (i.e., our AI models may produce either false positive results or false negative results), so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

Frequently Asked Questions

How should I interpret Turnitin's AI writing percentage and false positives?

The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI paraphrase tool or word spinner.

False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

What does 'qualifying text' mean?

Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Figure 6: AI Detection Image 1