Testify.ai

Group Members

Saad Ilyas (01-131222-041)

Basit Ali (01-131222-013)

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: __Saad Ilyas______________ Enrolment No: _01-131222041__

Student Name: __Basit Ali_______________ Enrolment No: _01-131222-013__

Programme of Study: Bachelor of Software Engineering

Project Title: ____Testify.ai____

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 plagiarism test of this thesis using HEC prescribed software and found similarity index at ___________ 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: ____16 April 2026___________ Name: ___Dr Tamim Ahmed Khan_____________

CERTIFICATE OF ORIGINALITY

This is certify that the intellectual contents of the project **Testify.ai** 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 it shall 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 authorities the University to cancel my/our degree.

Name of the Student: __Saad Ilyas________________

Signature: _______________________ Date: _16 April 2026______

Name of the Student: ___Basit Ali_________________

Signature: _______________________ Date: __16 April 2026______

PROJECT TITLE (MENTION PROJECT TITLE HERE)

Sustainable Development Goals

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

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			
	Attribute	**Complex Problem**	
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			
	Attribute	**Complex Activities**	
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.	

Abstract

This project presents **Testify AI**, an intelligent desktop-based software testing system developed to automate web application testing through artificial intelligence and browser automation. The main objective of the project is to reduce the time, effort, and complexity involved in manual testing by automatically analyzing a target website, identifying interactive user interface elements, generating relevant test cases, and executing them in a structured way. The system uses a combination of tools to build the desktop front end using Electron and angular. In addition, it is using a browser-based automation tool, called Playwright, to interact with browsers in order to create automated test cases. Additionally, the system includes a full ai-powered extraction & reasoning engine that analyzes web pages to identify page layouts, logically groups UI elements into user flows, and generates test cases (both static and dynamic) representing the various paths a user may take including positive & negative test case examples. This system will enable users to input their own user inputs at runtime for sensitive data (such as login credentials), evaluate the test output against expected behaviors/failure modes, show what was navigated during testing via visual representation of all page transitions occurred during testing process, and export the results of the tests into report formats for review or other use.

Keywords: Artificial Intelligence (AI), Automated Software Testing, Desktop Application, Electron, Angular, Playwright, Test Case Generation, UI Extraction, Browser Automation, Test Coverage, Quality Assurance, Human-Computer Interaction.

Dedication

First and foremost, we dedicate this work to Almighty Allah for granting us the strength, patience, and wisdom to successfully complete this project.

We sincerely dedicate this project to our respected supervisor, **Dr. Tamim**, whose invaluable guidance, continuous support, and insightful feedback were instrumental throughout the development of this work.

We also dedicate this achievement to our parents and families for their unwavering encouragement and prayers. Finally, we acknowledge **Basit Ali** and **Saad Ilyas** worked hard as a team to make this project a reality. Their shared focus and effort were the main reasons we were able to complete everything successfully.

Acknowledgments

We would like to thank God (Almighty) for giving us the chance to successfully finish our Final Year Project.

In addition to thanking God for allowing us to finish the last year of school, we would like to give credit to Dr. Tamim for his constant help throughout the entire process of creating this project. He was always available for us to ask questions, get advice from him when needed, and he helped guide us along the way so that we could stay focused and continue improving the quality of our work.

The Department of Software Engineering has given us the foundation of knowledge and environment to learn in. The availability of lab equipment, computers and administration has been invaluable in assisting us with finishing our project.

Also, we would like to thank the communities who have developed and supported the software platforms such as Electron, Angular, and Playwright. These programs gave Testify AI a good base of technology to build our program off of, and they made it much easier for us to document them. This has allowed us to make building our program and troubleshooting bugs much simpler.

Lastly, we would like to say thank you to our parents and families for their continual prayerful support and patience with us while working on this difficult task. Their support and encouragement allowed us to maintain the motivation needed to reach the end of this long journey.

Table of Contents

FYP Completion Certificate **1**
Certificate of Originality **3**
Project Title (mention project title here) **4**
Abstract **6**
Dedication **7**
Acknowledgments **8**
Table of Contents **9**
List of Figure **12**
List of Tables **13**
CHAPTER 1 ..... **14**
INTRODUCTION **14**
Introduction **15**
1.1 Motivation 15
1.2 Objectives 16
1.3 Maior contributions 17
1.4 Creating A Consistent Format For The Report 18
BACKGROUND STUDY / LITERATURE REVIEW **20**
Background Study/Literature Review **21**
2.1 Introduction 21
2.2 Traditional Software and Web Application Testing 21
2.3 GUI-Based and Model-Based Testing 22
2.4 Browser Automation Frameworks 23
2.5 Commercial AI-Based Testing Platforms 24
2.6 Model-Based Testing in Practice 27
2.7 Intelligent Exploration and Reinforcement Learning Approaches 28
2.8 Large Language Models in Software Testing 28
2.9 Relation of Existing Work to the Proposed Project 31
2.10 Chapter Summary 32
System Requirements **35**
3.1 Use Case Diagram 35
3.2 Functional Requirements 36
System Design **51**
4.1 Design Constraints 51

4.2 System Architecture 53
4.3 Logical Design 55
4.4 Dynamic View 56
4.5 Component Design 64
4.6 Data Models 66
4.7 User Interface Design 67
4.8 System Prototype 70
4.9 Conclusion 70
System Implementation 72
5.1 Introduction 72
5.2 Development Environment and Technologies 72
5.3 Overall Implementation Strategy 73
5.4 Frontend Implementation 73
5.4.1 Home Interface 73
5.4.2 System Testing Interface 73
5.4.3 Chat and Interaction Panel 74
5.4.4 Coverage Visualization Interface 74
5.5 Electron-Based Desktop Integration 74
5.6 Web Page Extraction Implementation 75
5.7 Test Case Generation Implementation 75
5.7.1 Static Test Generation 75
5.7.2 Dynamic Flow-Based Test Generation 76
5.8 Runtime Input Handling 76
5.9 Test Execution Implementation 77
5.10 Test Evaluation Implementation 77
5.11 Repository and Coverage Implementation 78
5.11.1 Test Case Repository 78
5.11.2 Coverage Manager 78
5.12 Export and Reporting Implementation 78
5.13 Integration of System Modules 79
5.14 Implementation Challenges 79
5.15 Conclusion 80
System Testing & Evaluation 82
6.1 Test Strategy 82
6.2 Component Testing 82
6.2.1 Frontend Interface Component 83
6.2.2 Electron Communication Component 83

6.2.3 Extraction Component 83
6.2.4 Test Generation Component....83
6.2.5 Execution Component....83
6.2.6 Coverage and Export Component....84
6.3 Unit Testing....84
6.4 Integrated Testing....84
6.5 System Testing....85
6.5.1 Functional Validation....85
6.5.2 Usability Validation....85
6.5.3 Reliability Validation....85
6.5.4 Performance Validation....86
6.6 Test Cases....86
6.7 Results & Evaluation....89
6.8 Conclusion....91
Conclusion 93
7.1 Contributions....93
7.2 Reflections....94
7.3 Future Work....95
Appendix A....99
Appendix B....99
Appendix C....99
Appendix D....100
Appendix E....100
Appendix F....101
Appendix G....102

List of Figure

Figure 1: Chapter Thesis 19
Figure 2: Use case of Testify.ai 36
Figure 3: Component Interface of Testify 39
Figure 4: Start Testing Session Sequence Diagram 44
Figure 5: Test case Generation Sequence Diagram 45
Figure 6: UI Extraction Sequence Diagram 45
Figure 7: User Input Resolution Sequence Diagram 46
Figure 8: Test Execution & Evaluation Sequence Diagram 46
Figure 9: UI Extraction Activity Diagram 47
Figure 10: Start Testing Session Activity Diagram 47
Figure 11: Testcase Generation Activity Diagram 48
Figure 12: User Input Resolution ActivityDiagram 48
Figure 13: Domain Model of Testify 49
Figure 14: System Architecture of Testify 55
Figure 15: Domian Model Diagram 56
Figure 16: Main Testing Pipeline Sequence Diagram 58
Figure 17: Execution Process Sequence Diagram 59
Figure 18: LLM Testcase Generation Sequence Diagram 60
Figure 19: Test Execution Sequence Diagram 61
Figure 20: Main Testing Activity diagram 62
Figure 21: UI Extraction Sequence diagram 62
Figure 22: LLM Test Case Generation Sequence diagram 63
Figure 23: Test Execution Sequence diagram 63
Figure 24: Component Diagram 65
Figure 25: Package Diagram 66
Figure 26: Deployement Diagram 67
Figure 27: Login page of Testify 68
Figure 28: Dashboard Page 69
Figure 29: Search Engine Page 69
Figure 30: Execution page 70

List of Tables

Table 1: Comparision with AI-Based Testing Platforms 26
Table 2: Evaluation of LLMs in Testing 30
Table 3: Relation of Existing Work 31
Table 4: Constraints 41
Table 5: Legal and Ethical Feasibility 43
Table 6: Website Analysis and Test Generation 86
Table 7: Runtime Credential Input and Test Execution 87
Table 8: Coverage Graph Generation 88
Table 9: Export of Test Results 88
Table 10: Functional Requirement Traceability Table 100

CHAPTER 1

INTRODUCTION

Introduction

Although, currently, there are many automation tools for testing available, most of them require developing your own test scripts which means if a website's layout changes the test script(s) will be broken.

To alleviate this issue, we created Testify AI. It is a desktop application utilizing Artificial Intelligence (AI), along with Browser Automation, to conduct UI extraction by identifying all of the interactive parts of a website, create automatic test cases from identified interactive sections of the website, execute those test cases through a predefined workflow to display results of executing the test cases, and also provide examples of what was tested.

You can export the test cases into a report at any point.

We developed Testify AI as a desktop application utilizing Electron, Angular, and Playwright. The primary focus of our build process was building the application around artificial intelligence-based logic to enable the system to understand different interface formats. Our goal is to decrease the amount of manual work required during testing. And, we feel that by increasing automation and making it smarter, we can improve overall software quality without adding additional man-hours.

As long as users continue to develop websites that include interactive elements such as buttons, links, dropdown menus etc., they need to continuously test their sites and ensure that these interactive elements continue to work as intended.

1.1 Motivation

Our reason for pursuing our idea for this project was due to the growing complexity and difficulty of testing new Web Applications. Manually testing the entire functionality of a given application takes far too long and requires far too many resources. When you have to click, enter data into forms, and go back and forth to the same page over and over again to perform one test case it becomes even worse. Manual testing is both time-consuming and cumbersome. Additionally, the potential

for human error exists, which may lead to missed defects or reduced accuracy of results.

There are numerous automated tools that exist to aid in automating your tests. However, most of them require writing and maintaining test code by hand. What makes these tools difficult to use is that they typically require programming knowledge. Even if you do possess some coding skills, every time a website experiences changes to its layout you will need to update your test scripts. Therefore, we required something better; something that would provide as much automation as possible in the manual testing process. We wanted to increase the rate at which we could test and increase the coverage of our testing.

Therefore, our focus for this project was creating an application that utilizes artificial intelligence along with web-browser automation to develop a system that learns about web-pages itself. The application should generate and execute test cases with a minimum amount of interaction from the users. Our overall goal was to make software testing faster, less expensive and more accessible through the creation of a smart desktop application that improves the quality of software. Ultimately, this project provides solutions to common problems experienced within software engineering, and shows just how significantly AI can contribute to improving Automated Software Testing today.

1.2 Objectives

The goal of this project is to build a Smart Desktop Application using AI to automate the generation of test cases from Web Applications. This will utilize Electron and Angular technologies and ultimately increase efficiency when it comes to testing Web Applications.

The objectives of this project are:

1. Develop a desktop application for automating the testing process for software applications.
2. Identify all possible User Interface (UI) elements on every Web Application page to extract all possible user interactions.

3. Automatically create test cases for user journeys that may consist of both basic and advanced navigation paths.

4. Use Browser Automation Tools like Playwright to execute the created test cases.

5. Determine if the tested Web Application executes as expected or fails during the execution of the test cases.

6. Compare the results of actual behavior vs expected behavior during test case execution to determine if the tested Web Application works as intended and identify any potential issues.

7. Evaluate Test Results based on Expected Behavior & Failure Criteria.

8. Create a visual display identifying the Web Application Pages that were part of the test plan along with the paths taken to reach those pages.

9. Provide End Users with a way to Save / Export their Generated Test Reports in File Formats supported by most Reporting Packages found in Business Environments.

10. Reduce the amount of time Developers normally spend writing out Test Script Code manually.

11. Leverage Artificial Intelligence (AI) Technology to Automate/Simplify/Facilitate Improvements to Quality Assurance Processes for software applications.

1.3Maior contributions

1. We have created a desktop application which utilizes artificial intelligence (ai) and browser automation technologies combined together in one single platform.
2. Within this desktop application is a smart workflow engine that can automatically extract ui elements from websites, and generate test cases for automated software testing without requiring the user to handcraft every script for each website and/or test case.
3. Our solution offers static testing capabilities on individual elements, and dynamic testing capabilities on full user flows making it a more practical solution for the process as a whole.
4. Our system can automatically execute, and evaluate test cases to determine if the app's behavior meets expectations quickly.
5. In addition to reporting test coverage, our solution also generates visuals depicting how well pages are being tested within the application, and provides export functionality for reporting to document, track progress of testing efficiently.

6. Also included with our solution is the ability to enter sensitive information, such as login credentials at runtime so users can perform real-world testing.

Significance of the Project

This project has potential for use as it will eliminate the time required for all teams to write every single test script individually. Since the new desktop application will allow users to automate software testing at a much faster rate, using less resources than they did previously, the team will have fewer hours dedicated to QA. The Desktop Application was designed to be adaptable which is a huge advantage given how quickly web applications are evolving and layouts/functionalities can change often. In addition we streamlined many aspects of professional testing such as UI extraction and test case creation which will give users (testers) the ability to perform these types of tests even when there are multiple updates made to their website.

What Is Better Because of This Project

The benefits of this project are numerous;

This Project enables a much better use of Artificial Intelligence (AI) in combination with Intelligent Web-based Testing, while minimizing the amount of Manual Labor required.

In addition, This Project will accelerate Test Case Generation and Execution on Real World Application Flow using the Playwright.

Further, This Project allows for the automation of Page Scanning and UI Extraction to track Navigation Pathways to produce higher levels of Accuracy than the previous methods were able to accomplish.

Finally, AI is a valuable tool that can enhance the Efficiency of Automated Software Testing, Quality Assurance, and the overall Software Development Process in Desktop Applications.

1.4 Creating A Consistent Format For The Report

We plan to break up the report into six different areas or sections so we can make it easier for you to find whatever information you may want about your project. Each of the above areas will be identified by chapter number (Chapters 1-6) and each area will contain specific types of information.

The first Chapter covers the history of your project and will outline the main objectives of the project and give you an idea of all the work accomplished during the course of your project.

Chapter 2 will go over other previous studies that were done with regards to automation testing of applications via web browsers utilizing tools built on Artificial Intelligence. We feel like going over previous studies will help you understand what else has been done in the same area as your project.

Chapter 3 will describe how we designed the architecture for Testify AI Desktop Application.

Chapter 4 will explain both of the most important features that Testify AI Desktop Application has and show how those features operate together in a process.

Chapter 5 will present the results from our test runs of Testify AI Desktop Application. Those results will visually represent which pages were covered and also give us an indication of whether Testify was successful at completing its task.

Chapter 6 will summarize all of our results from this report and identify some areas where Testify could use further improvement. Also, we would like to suggest several ways we believe you can expand upon Testify after reading this report.

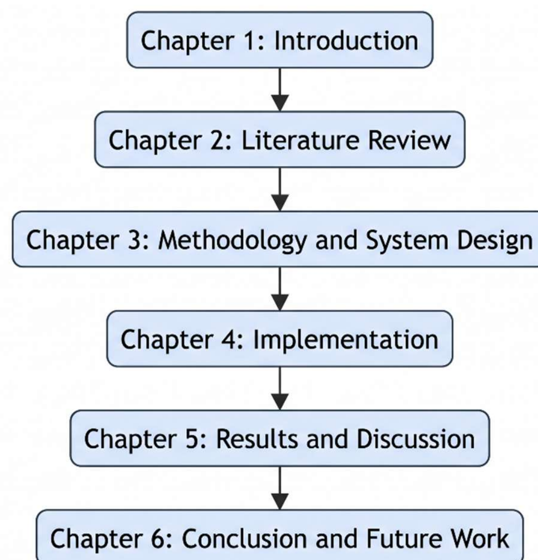

Figure 1: Chapter Thesis

CHAPTER 2

BACKGROUND STUDY / LITERATURE REVIEW

Background Study/Literature Review

2.1 Introduction

Automated software testing provides developers with a method by which the functionality and dependability of their applications may be assured. However, automated testing of web applications has become increasingly difficult because of the evolving characteristics of modern websites and the interactive nature of those sites. Manual testing of web applications is often time-consuming and labor-intensive. Therefore, numerous approaches to automated software testing have been created. This chapter describes the background and related work to our research project. In particular, this chapter addresses the use of traditional automated testing tool, graphical user interface (GUI) based testing techniques, artificial intelligence (AI)-based test automation solutions, and the application of large language models to generate Test Cases. The primary goal of this literature review is to illustrate how previous technologies support test development, identify shortcomings or limitations in prior technologies that could be addressed through new technologies like Testify AI, and clearly demonstrate why a desktop application like Testify AI was needed.

2.2Traditional Software and Web Application Testing

Testing can be categorized into two main types of testing - Automated testing and Manual testing. Manual testing relies on humans to determine if a particular System is functioning correctly. Although, a major drawback of manual testing is that the method is often slow and repetitive which can cause some possible errors to go undetected. On the opposite side of things, Automated testing provides faster and more consistent testing results, but usually requires developers to write their own test script and then continuously maintain those test scripts so that they continue to identify problems within each component of the System.

Many studies show that when performing tests on Web applications, most current testing approaches rely heavily on Functional testing (testing how a System operates), Navigation testing (testing how users navigate a website), etc. Although Functional/Navigation testing is effective at finding defects, it is designed to work with a static site layout and/or user flow. As most sites are constantly being updated

(new layouts, new user flows, etc.), assuming that a site will always operate under the same conditions is unrealistic.

Traditional testing methods provide reliability to known processes, its largest shortcoming is that it takes a large amount of time to build and repair test cases manually. It was due to these issues with the manual development/test case corrections, that led us to start this project. We wanted to find ways to automate the manual process of developing/testing cases using UI extraction techniques to allow the automatic generation of Test Cases based on interface identification.

2.3GUI-Based and Model-Based Testing

Studies about testing Graphical User Interfaces (GUIs) has been happening for years. Banerjee et al. did a study about the studies done in this field. Their results show that there are many studies about GUI testing. Some of the things those studies were looking at are: event-based testing; model-based testing; regression testing; and automated exploration. Even though their data shows that the foundational pieces for GUI testing are established, they still see challenges like scalability, automation, and maintenance as being long-term issues.

Memon, Pollak, and Soffa were among the first to do research to help improve the area of GUI testing. They looked into using automated planning to create hierarchical structures for GUI test cases. What they essentially did was view the user interface as a structure in which interactions occur, and then created test cases by determining every possible sequence of actions. The importance of this is that it took the focus off of randomly clicking at various parts of an interface and placed emphasis on creating tests where each one had a specific objective.

One major contribution to GUI testing with automated techniques came from Memon, Banerjee, and Nagarajan. They introduced the technique of "GUI Ripping," and showed how it used automated navigation to travel through an interface, how it would extract widgets and other interactive components from the interface, and how it would generate models for future test case development. The basic premise behind this concept is similar to what we're going to do today with Testify. Testify develops tests

after identifying the interactive elements in a target application's interface and creating a series of tests.

While many of the previously mentioned methods represent advancements in testing compared to non-automated testing methods, they also have limitations. Many of the previously discussed approaches were designed for testing against static desktop applications or static interfaces. Web applications today are much different than either type. Today's web applications rely heavily on client-side rendering. They can experience runtime state changes. And most web applications employ asynchronous behavior. While many of the concepts used in previous methods (like GUI ripping and model-based testing) can be adapted to work with modern web applications, modifications will likely need to be made.

2.4Browser Automation Frameworks

Automated testing options for browsers have become one of the more common ways to evaluate whether or not web-based applications are working properly. Of all of the automated testing tools available for evaluating whether or not web-based applications are functioning properly, there may be no tool more well known than Selenium. Selenium enables users to utilize WebDrivers to control browsers in much the same manner that they are controlled by humans when they manually interact with them. With this flexibility comes the possibility of utilizing Selenium as a powerful and flexible platform for simulating how users might interact with websites using various types of browsers. Although, while Selenium is a flexible platform for performing automated testing; it does not provide the necessary guidance or structure needed to write tests. Because of this; the responsibility for designing and maintaining tester locator strategies, executable actions and assertions fall directly on the testers themselves.

Cypress was created with the intention of providing a new, unified testing paradigm for end-to-end and component/unit testing. By virtue of its design, Cypress makes debugging easier; keeps tests transparent and accessible to developers; and allows developers to complete tasks rapidly and easily. As a result, Cypress is exceptionally well-suited for modern front-end application development. Like Selenium; however; Cypress requires developers to create the actual tests. While Cypress will certainly

help streamline the process of executing tests; Cypress will not be able to determine what needs to be tested until the developer has first identified those items and executed the associated tests.

Playwright continues the trend of enhancing previous browser automation functionality with additional features such as auto-waiting (which reduces the need for artificially-added waiting times) and the capability to run tests on multiple browsers concurrently. Additionally; Playwright runs each test in a separate environment and utilizes a feature referred to as "test generation"; which is nothing more than a code-generating tool that assists developers in writing tests. One of the primary features of Playwright is the capability to record your actions taken against a website and generate high-quality locators. The benefits mentioned above have made Playwright a preferred option for consistently executing browser-based tests. For this reason alone, Playwright is being utilized as the primary execution engine in Testify.
Although these frameworks are powerful, their main limitation is that they remain execution-oriented rather than intelligence-oriented. They are excellent at running tests, but they do not independently understand the page, discover meaningful flows, or reason about test intent in a human-like manner. This gap is one of the central problems addressed by Testify AI.

2.5 Commercial AI-Based Testing Platforms

I see that many of today's Industrial Applications have utilized artificial intelligence to address many issues related to scripting-based testing. Testim is a great example of a testing tool based on AI which will allow for less maintenance using "smart" locator technology and self-healing. The greatest benefit of Testim is that it makes it much easier to avoid breaking automated test scripts by making changes to the User Interface (UI) compared to other tools. A major disadvantage of Testim is that because Testim is a commercial product there is very little exposure into the actual workings of the software. Additionally, there are few opportunities to manipulate the code academically.

Mabl is another popular platform to utilize for testing purposes. Mabl uses a combination of Browser Automation, Artificial Intelligence to build and repair Tests,

and Full Web, API, and Performance Testing. Additionally, Mabl provides businesses a wide array of options for automating testing with minimal development time needed to write tests. Therefore, the primary value proposition of Mabl is how easy it is to use in a real-world industry environment. Unfortunately, the workflows provided by Mabl lock users into utilizing the Mabl platform exclusively, thus preventing users from having full control over their testing pipeline.

Applitools was a pioneer in Visual Testing but has expanded to provide autonomous testing capabilities. Applitools provides a system capable of exploring websites; building tests; and visually identifying any changes made to the UI. Visual testing can be a valuable asset in finding UI regressions that standard functional checks often miss. A potential drawback is relying solely on visual testing does not always provide the true intent behind a flow or scenario that the user intended to accomplish.

In addition, Cypress is developing artificial intelligence support via Cypress Studio AI which suggests assertions to the user while recording actions. Many of the larger testing platforms are beginning to incorporate artificial intelligence into their products. Although some products are offering AI assistance, the majority still operate as standalone utilities with only partial automation. While these tools are intelligent enough to understand single steps in a workflow and map out coverage, they are not yet smart enough to determine behavior and develop complete test scenarios all within a single unified application.

When reviewing all of the examples illustrated above, we see that while commercial platforms are generally excellent at ease of use and resilience, they generally act as ecosystems. On the other hand, Testify provides a testing workflow that is far more transparent than the closed systems listed above; provides better opportunity for academic study of the workflow; and is completely customizable.

Table 1: Comparision with AI-Based Testing Platforms

Tool / Platform	Type	Key Features	Strengths	Limitations	Relation to Testify AI
Selenium	Traditional Automation	Cross-browser testing, supports multiple languages	Highly flexible, widely used	Requires coding, no AI support	Base inspiration for automation
Cypress	Modern Framework	Fast execution, real-time reload, JS-based	Easy debugging, developer-friendly	Limited cross-browser support	UI interaction inspiration
Playwright	Modern Automation	Multi-browser support, auto-waiting, parallel execution	Reliable, powerful selectors	Requires scripting knowledge	Used in Testify AI backend
Testim	AI-Based Tool	AI-based test maintenance, smart locators	Reduces maintenance effort	Paid tool, limited control	Shows AI automation trend
mabl	AI Testing Platform	Auto-healing tests, cloud execution	CI/CD integration	Subscription-based	Similar automation vision
Applitools	Visual AI Testing	Visual regression testing using AI	Accurate UI validation	Limited to visual testing	Inspiration for UI validation
Cypress Studio AI	AI-Assisted Testing	Record and generate tests automatically	Easy test creation	Limited flexibility	Similar to Testify AI concept

2.6Model-Based Testing in Practice

The use of models has proven to be one of the most popular testing methodologies (and also is widely recognized as an acceptable methodology), both in academic settings and in the high-technology arena. Models provide a method for organizing complex systems into a structured format, allowing them to be tested in a systematic manner. According to research completed by Garousi et al. (2016), using model-based testing on mobile and web application systems produce quantifiable improvements such as better branch coverage rates, better test quality ratings overall, and substantially improved defect detection rates. Therefore, these empirical findings demonstrate that modeling remains very applicable for testing large/complex systems.

While this type of testing does have its advantages, it also has some drawbacks. A major disadvantage in developing, updating, and maintaining the models themselves is the cost involved in performing these tasks. In addition, if the applications being considered change rapidly over time, then the cost of producing/updating/maintaining models increases dramatically in terms of dollars spent and time consumed. As an example, many of today's modern web systems generate their user interfaces dynamically at run-time and users continually modify their workflows based upon the dynamic nature of the current system state.

According to Brunetto et al., they examined methods for implementing automatic test case generation within a production environment. Brunetto et al. stated that while this methodology may appear attractive from a theoretical perspective, successful implementation of this methodology requires careful evaluation in terms of the scalability of generated tests, the utility of generated tests and reporting test results. Thus, we currently have similar goals for our ongoing projects. Like other automated testing tools available today, Testify automatically develops test cases, executes those test cases, measures whether or not each test case resulted in either expected results or unexpected results and reports the results in a way which makes the results easy to interpret.

2.7 Intelligent Exploration and Reinforcement Learning Approaches

Many researchers are working on "Smart Exploration" as one area of focus for web testing. Rather than simply reusing common scripts for testing, several researchers are looking at ways to develop new methodologies that can produce useful sequences of interactions by autonomous discovery of an application's behavior. Chang et al proposed a methodology called Web QT (Web-based Quality Testing), which employs reinforcement learning to automatically generate test cases for web based applications. Web QT identifies the best sequence of user interactions to achieve maximum code coverage during automated discovery.

The primary advantage of utilizing reinforcement learning is its ability to be adaptable. Reinforcement learning allows intelligent navigation through an application much better than simple random clicking or basic web crawling. There are however disadvantages to this type of navigation. Simply achieving a high code coverage count does not necessarily mean you've created good quality tests. For instance, if a bot navigates a large volume of pages yet fails to identify the most critical user activity to understand the underlying logical flow of the application, it has failed at providing good quality testing. Critical business processes such as login, signup, payment processing and form validations require the system to understand the underlying logical flow of the process versus performing blind exploration.

Again as referenced above, we are currently addressing the exact problem Testify does not use random navigation throughout the entire site nor does it pursue a high level of code coverage. Instead, Testify analyzes interface behaviors and uses AI reasoning to find relevant user flows used for testing purposes and creates test case scenarios with meaning.

2.8 Large Language Models in Software Testing

The advent of Large Language Models (LLMs) enables us to approach testing of Software Applications in a totally new way. With capabilities to read natural language, extract data from user interface descriptions, generate test cases and assist with the creation of a test oracle, we can now observe the expanding presence of AI

based methodologies for GUI Testing which have been developed utilizing NLP/RL and begin to see LLM's used to produce test cases and support the overall testing process.

As well as showing how we can employ Large Language Models to create test cases researchers demonstrated that LLMs allow for faster production of test cases, reduce the human involvement needed to produce them and generate a wider variety of test case scenarios than would typically be produced manually. Researchers did however identify several obstacles related to employing LLMs in this fashion. These included variation in both the output quality and consistency of the model, dependence on the quality of the prompt(s) being employed and difficulties in determining if the generated test cases met the specific needs and objectives of the project. While there appears to be considerable opportunity to use LLMs for testing purposes, it should be carefully integrated into existing testing architectures.

Another researcher has more recently reported his conclusions regarding evaluation of LLMs within the context of the larger testing paradigm. The researcher emphasized that LLMs need to be evaluated against high-quality benchmarking data sets; compared fairly to test cases created by humans; and subjected to realistic simulation of actual usage. The feedback provided by this researcher is important. It clearly indicates that relying solely on AI-based solutions for creating test cases does not suffice because successful execution is not enough; the test cases must also successfully validate and have practical value.

The Researcher, Le et al. was able to combine screen transition graphs and LLMs to achieve end-to-end web application testing. The Researcher showed that when combining structural navigation models with the logic of LLMs produces a very effective solution. This methodology supports the core design philosophy behind Testify. The Testify pipeline effectively combines the extracted UI structure, defined user flow and AI-generated test cases to function as a single unit.

Table 2: Evaluation of LLMs in Testing

Study	Year	Main Idea	Technique Used	Strength	Limitation	Relevance to Testify AI
Dogan et al.	2019	Automated test generation	Model-based testing	Reduces manual effort	Limited scalability	Base for automation
Banerjee et al.	2020	Intelligent UI testing	Machine learning	Adaptive testing	Needs training data	AI-based testing idea
Memon et al. (Planning)	2001	GUI test planning	Event-based testing	Structured approach	Complex implementation	GUI workflow inspiration
Memon et al. (GUI ripping)	2003	Extract UI structure	GUI ripping	Automatic UI extraction	Limited dynamic handling	Similar to DOM extraction
Garousi et al.	2018	Systematic testing review	Literature survey	Comprehensive study	No implementation	Research foundation
Brunetto et al.	2008	Web testing automation	Crawling-based testing	Good coverage	Limited intelligence	Navigation testing idea
Chang et al.	2021	AI-driven testing	Deep learning	Smart test generation	High complexity	AI inspiration
Amalfitano et al.	2015	Mobile GUI testing	Crawling & exploration	Automated exploration	Platform-specific	Exploration idea
Tasarsu et al.	2022	Reinforcement learning	RL-based agents	Adaptive learning	Slow training	Smart navigation

		testing				
Li et al.	2023	LLM-based testing	NLP + LLM	Natural language testing	Accuracy issues	Core idea of Testify AI
Le et al.	2023	Test generation using LLM	GPT-based models	Easy test creation	Needs prompt tuning	Direct inspiratio n

2.9 Relation of Existing Work to the Proposed Project

Table 3: Relation of Existing Work

Existing Approach	What It Does Well	Limitation / Gap	How Testify AI Addresses It
Traditional automation tools	Reliable test execution	Requires manual scripting	Uses natural language input
GUI/model-based testing	Structured testing flows	Complex setup	Automates flow generation
Commercial AI platforms	Smart automation	Paid and limited customization	Custom AI-driven system
Reinforcement learning approaches	Adaptive behavior	High training cost	Faster AI-based decisions
LLM-based testing approaches	Natural language support	Accuracy issues	Combines LLM + DOM extraction

We found through my own research that there is no single tool that can successfully develop and implement "smart" automated web testing. Tools such as Selenium, Cypress, and Playwright have very successful results with regards to automating browsers; however, they are limited to how much development work is required of the developer to create the individual test cases. While the two other approaches mentioned (GUI Ripping & Model Based Methods) provide the necessary structure to

automate test case creation, both methods fall short when applied to modern web applications which continue to become increasingly dynamic. While commercial AI tools provide significant time savings in terms of maintaining an application under test, many require a subscription or license agreement to be able to access its functionality. Furthermore, most AI based testing solutions are designed around platform specific testing solutions and lack the flexibility needed for open-source academic research. Many recent advancements utilizing Large Language Models show great potential; however, nearly all focus on developing test code and none of the existing LLMs currently used in automated testing perform the execution of the generated test code nor report the final results.

Our proposed solution Testify takes the best aspects of each of these currently available tools and remedies their respective shortcomings. As with GUI-Ripping methods, Testify extracts interactive UI components directly from the webpage. As with traditional automation frameworks, Testify performs actual click actions and interacts with elements within the browser. Additionally, as with modern AI systems, Testify utilizes intelligent decision making to generate meaningful test cases. What truly differentiates Testify is how we utilize all of these features into one complete desktop application. With this design, Testify is capable of analyzing the user interface, creating user flow diagrams, executing both positive and negative tests, receiving input from users during runtime, comparing expected vs. actual results, and providing a visual representation of test coverage.

Due to this feature set, our proposed project is not simply another concept floating in a void. Rather, it is a practical, hands-on methodology that allows us to combine the most viable principles of GUI Testing, Browser Automation, Model-Based Testing, and AI driven Software Engineering into a singular cohesive system.

2.10 Chapter Summary

We will conclude our coverage of the foundational background information related to this project. In this chapter we have thoroughly covered standard web testing, GUI-based testing, commercial browser-automation products, commercial artificial intelligence testing products, model-based testing, reinforcement learning and the use

of large-language-models for testing. Clearly, each of the above listed areas of technology has tremendous potential; however, all of them currently fall short because none can fully automate every possible system (i.e., there are always exceptions), none can be truly adaptive when faced with constantly changing systems, none can generate fully transparent reasons for why tests fail (or otherwise produce undesirable results), and none can be easily integrated with other types of software applications.

Each of the listed areas' shortcomings were the driving force behind developing Testify. Testify was developed as a desktop focused application (intelligent) which would automatically discover all interfaces of a given website or web application, create and execute test cases from those discovered interfaces, analyze the output from each executed test case and evaluate the percentage of total test coverage achieved against one complete system. The next chapter presents the methodology and system design adopted to implement this solution.

CHAPTER 3
SYSTEM REQUIREMENTS

System Requirements

This chapter presents the system requirements of **Testify AI**, an intelligent desktop-based web testing application. It defines the required system behaviour, interfaces, storage needs, quality attributes, feasibility considerations, and analysis models. The purpose of this chapter is to specify how the proposed system operates, what services it provides to the user, and what technical and operational conditions are necessary for its successful deployment.

3.1 Use Case Diagram

The main actor of the proposed system is the User/Tester, who interacts with the Testify AI desktop application to perform automated testing of web applications. The user begins by entering a target URL, after which the system analyzes the web page, extracts interactive user interface elements, generates test cases, executes them, and displays results. During execution, the system may request runtime values such as credentials for input fields. After execution, the system presents test reasoning, coverage information, and exported results.

Follows are the use cases we included:

- Enter target website URL
- Start page analysis
- Generate static test cases
- Generate flow-based test cases
- Provide runtime credentials or required input values
- Execute generated test cases
- View AI reasoning and test results
- View coverage graph
- Export test report

.

3.1.1 UseCase Diagram

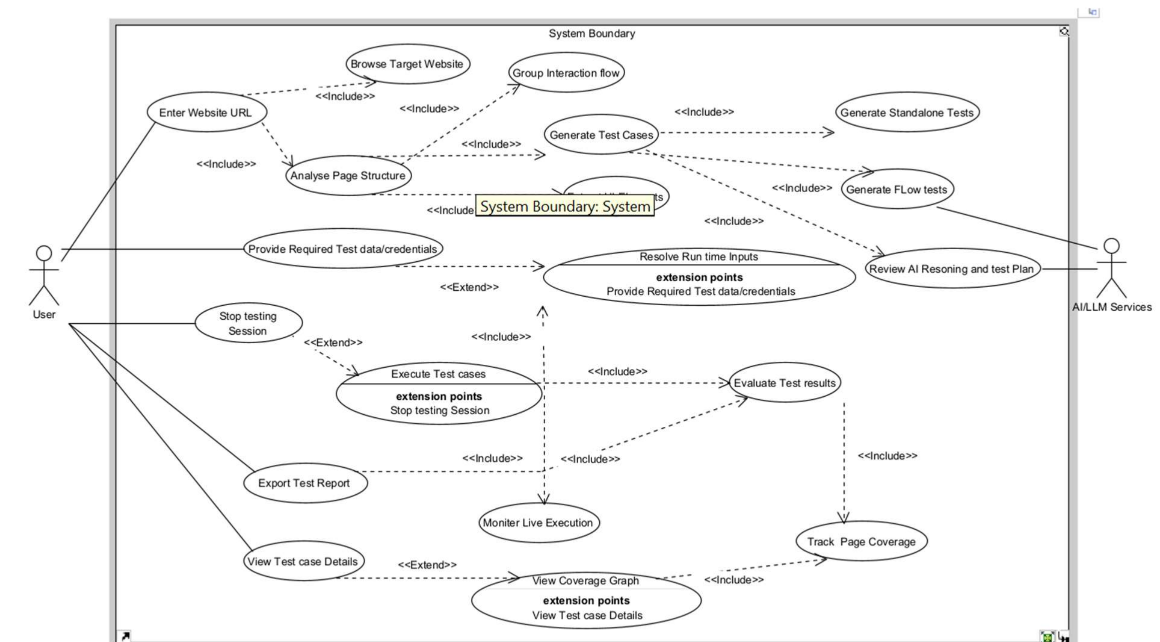

Figure 2: Use case of Testify.ai

3.2 Functional Requirements

3.2.1 Testify AI's functional requirements map out the core tasks the software is actually built to handle.

3.2.2 URL Submission

The System should allow the users to input a valid URL. It is serves as a main starting point of this Project.

3.2.3 Web Page Analysis

The system should analyses the web page given by the user and extract interactive elements such as buttons, links, text boxes, checkboxes, dropdowns etc.

3.2.4 Static Test Case Generation

The system should generates a static test case for standalone UI elements e.g. buttons and links. These tests should verify that the basic independent actions like clicking and navigation.

3.2.5 Dynamic Flow-Based Test Case Generation

You may create test cases that simulate typical user interactions (login/signup/submit form etc.) and dynamic routes.

Test cases can be created from both the positive path and negative path of potential user experiences.

3.2.6 Runtime User Input Handling

Any time a test requires the user to submit real-world data (username/password/etc.) the application will ask for this data at run-time.

3.2.7 Test Case Execution

Tests will be executed using automation tools.

Each test will act as though the user was manually navigating through a browser/website.

3.2.8 Test Result Evaluation

After running each test, your tool should determine if the actual results equate to the expected results.

As well as comparing expected results, your tool should check that any relevant error message were presented correctly.

3.2.9 Coverage Visualization

A visual representation of which pages were navigated and what routing options were used to help identify areas of the automated testing process.

3.2.10 Result Export

All test result data should be stored into a report/document in a clear format (Excel)

3.2.11 User Feedback and Reasoning

The end-user should receive notifications/status updates regarding every aspect of the testing process (data pull/write/test/run/results/etc.).

The user should be able to understand why specific actions were performed by the tool

3.3Interface Requirements

The interface requirements outlines what the interface for each component of this project needs to look like for end-users, i.e., how users interact with the program, how

the program interacts with its computer (physical hardware) and other software components.

3.3.1 User Interface Requirements

User Interface = An electron/angular based graphical user interface. A user opens the tool and first sees a "Home" Dashboard which provides the ability to select a testing module and enter a URL to begin. When you have selected both options, the screen splits into two main sections. The left side of the screen shows the browser view showing the web site(s) being tested. The right side of the screen displays a live feed of the systems' thinking and progress updates along with any prompts for password or further input from the tester. In addition to the browser and chat windows, the dashboard has a visual representation of where pages have been visited during the test using a "coverage map". As mentioned before, test results can be exported when you click on the "Export Results" button.

3.3.2 Software / Component Interface Requirements

In terms of behind the scenes, our application consists of multiple software applications communicating with each other. Our angular front-end communicates with the central Electron process via Inter-Process Communication (IPC). The Electron process acts as the brain of the operation and manages all of the actions, extracts data, coordinates with the AI to generate test scripts, runs the test, and reports back on the outcome. We utilize Playwright in order to simulate a human while moving around a web-site by generating mouse clicks. The AI and LLM modules do most of the heavy lifting for creating logical pathways out of elements extracted from the web-page and frameworks for creating test cases. While the tests are running, the repository and coverage modules hold all real time data/mappings that tie together all of the data-points. Once the test session is complete, the Export Module holds all final test case results in document form.

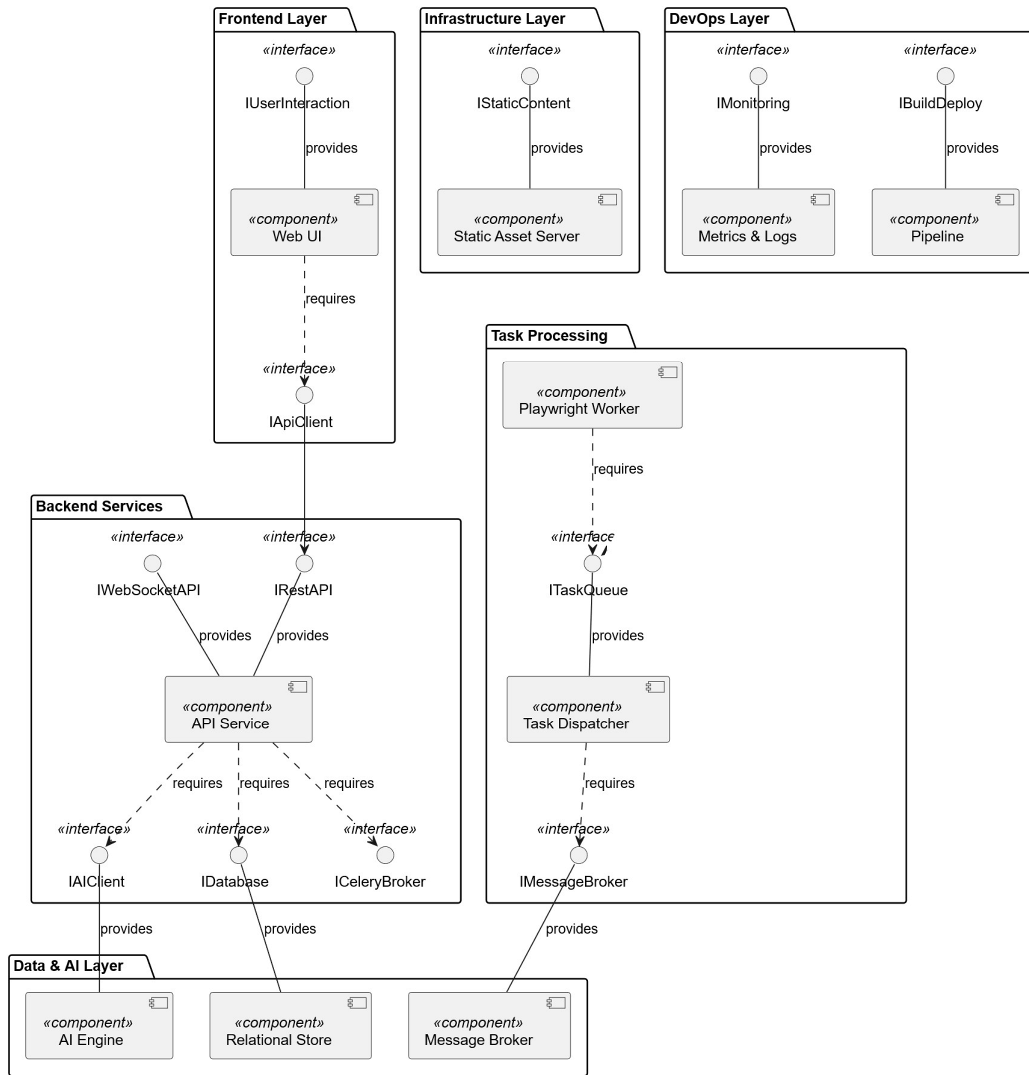

Figure 3: Component Interface of Testify

3.3.3 Physical Interface Requirements

User can run the system on any system that supports window/Linux. Internet Access is Compulsory.

3.4Database Requirements

The Proposed System Requires Data Base for User management & Admin Management because user Credentials like name, email etc and Users banking information like credit card Monthly token usage need to be Stored. Also DB requires for Admin Management like how many user, Api usage for specific user.

3.5Non-Functional Requirements

It defines the quality attributes and constraints of the system.

3.5.1 Performance Requirements

We expect the application to react quickly to user actions. When the user enters a valid web address the application will quickly start analyzing that webpage. Although the application will take some reasonable amount of time to create and run the tests — depending upon how complex the target webpage was — once all of the tests have finished, the coverage map and final test results will show up on the screen shortly after.

3.5.2 Usability Requirements

The product needs to be simple enough to use, so users can start testing right away without needing much training. All major elements of the product — which include entering a URL, starting tests, typing in passwords, etc. And displaying results — must be easy to find. In addition to giving you visual feedback about what is happening during each action of testing, the application will describe how it arrived at those decisions. Therefore, at all times during testing, users will always know what is going on “behind-the-scenes.”

3.5.3 Reliability Requirements

The tool cannot fail due to a dead link, failure to retrieve data or due to a runtime error. Rather than crashing in these situations, it should recover cleanly, and give users clearly-defined and actionable feedback as to why something did not work. The tool must perform consistently across the entire test cycle regarding whether a test passed or failed. There must be full consistency in the test flow from start-to-finish.

3.5.4 Security Requirements

The tool will only ask users for sensitive information, like login credentials, whenever they want to proceed with the next step of testing. No sensitive information collected by the tool will ever be stored. Additionally, the tool will only send communications back-and-forth with web sites using safe, pre-determined automated communication methods.

3.5.5 Maintainability and Modifiability

Since we're creating the system in modular units, developers can make modifications to individual components — i.e.: data extraction, test creation, AI processing logic, execution, or coverage maps — without affecting any of the other modules. The modular nature of this design enables us to easily add new functionality or updated technologies (i.e.: newer AI-based testing algorithms) into our products without having developers build everything again from scratch.

3.5.6 Interoperability Requirements

To enable our platform to operate effectively, it must combine several disparate technology stacks (i.e.: angular/electron/playwright/AI testing platforms), and work seamlessly with existing web applications and common browser formats.

3.5.7 Constraints

For proper operation, the application requires constant connectivity to the internet in order to test actual web sites and execute its AI processes. Pages that are dynamic and extremely hard to interpret, web sites that are heavily restricted by anti-bot security measures, locked-down web sites, and/or poor internet connections may cause issues related to performance of the application. Finally, the better quality of the tests that were created directly relates to how accurately the application can determine the initial structure/layout of the web site.

Table 4: Constraints

Category	Summary	Conclusion
Performance	Fast response and timely test execution with prompt result display.	Satisfactory
Usability	Intuitive interface with minimal training required.	High
Reliability	Graceful error handling and stable workflow.	Reliable
Security	Secure handling of credentials and authorized testing only.	Secure
Maintainability	Modular design enabling easy updates and extensions.	Maintainable

Interoperability	The integration of Angular, Electron, Playwright, and AI Components.	Compatible
Constraints	Need live internet Connection and depends on external websites' natural behavior.	Compatible

3.6Project Feasibility

3.6.1 Technical Feasibility

Technically, the completion of this project should be straightforward as we're using all standard, common and well understood tools to build it. The use of Angular gives us a solid base on which to create today's standards for our front end; and we can then embed everything into a single application for the desktop via Electron. Additionally, since we'll be using Playwright to automate the browser itself, I don't see where we'd run into trouble with that either. Lastly, we can access the AI component from our tests via API calls to retrieve the data, and since we'll be testing locally instead of using the cloud, we won't incur much if any expense or complexity when it comes to hardware needed to complete this project.

3.6.2 Operational Feasibility

From an operations standpoint, this application directly addresses one of the largest issues in software quality assurance that developers, test engineers and students face daily; i.e., eliminating the numerous hours required to conduct repetitive, manual testing.

All three user types (developers/testers/students) are able to utilize this single application for the automation of testing that previously consumed countless man-hours when performed manually.

The ease of the desktop version will allow both large teams and small development/quality engineering teams alike to set up and run the application easily.

3.6.3 Legal and Ethical Feasibility

The use of this tool to evaluate any web site that you are lawfully able to evaluate is completely legitimate and permissible from both an ethical standpoint and a legal one. Keep in mind that if you plan to utilize a script (automated) on another persons network, you will first have to obtain their approval. Be mindful of how you utilize password(s) that are logins to sensitive information. Do not attempt to bypass the

security that a platform has implemented via the use of an automation script, do not attempt to breach a platforms user privacy agreement, or violate the terms of service of a platform. Use the results provided by the AI as suggestions for evaluation purposes only. DO NOT consider anything provided by the AI as fact. Verify all results generated by the AI prior to implementing large scale changes based upon it.

Table 5: Legal and Ethical Feasibility

Type	Summary	Conclusion
Technical	Utilizes Angular, Electron, Playwright, and AI APIs; runs on Cloud System.	Feasible
Operational	Takes the manual work out of web testing, giving students and industry professionals a straightforward desktop app they can easily navigate.	Feasible
Legal & Ethical	Mandates that you only run tests on approved platforms, keeps private details secure, and makes certain the AI tools are handled responsibly.	Feasible

3.7Analysis Models

The analysis models will help explain the internal workflow of Testify AI and making it easy to see exactly how information travels across the different parts of the system.

An activity diagram can represent the overall workflow beginning from URL submission, followed by page extraction, UI grouping, test generation, optional credential input, test execution, result evaluation, coverage generation, and report export.

A sequence diagram can represent the interaction between the user, frontend interface, Electron controller, extraction module, AI/LLM module, browser automation engine, repository, and coverage manager. This shows how messages and outputs flow during a test session.

A data flow diagram or system flowchart can be used to illustrate how the input URL is transformed into extracted elements, then into test cases, then into execution results, and finally into coverage and exported reports.These models are useful because they present both the logical flow and the interaction flow of the system in a clear visual form.

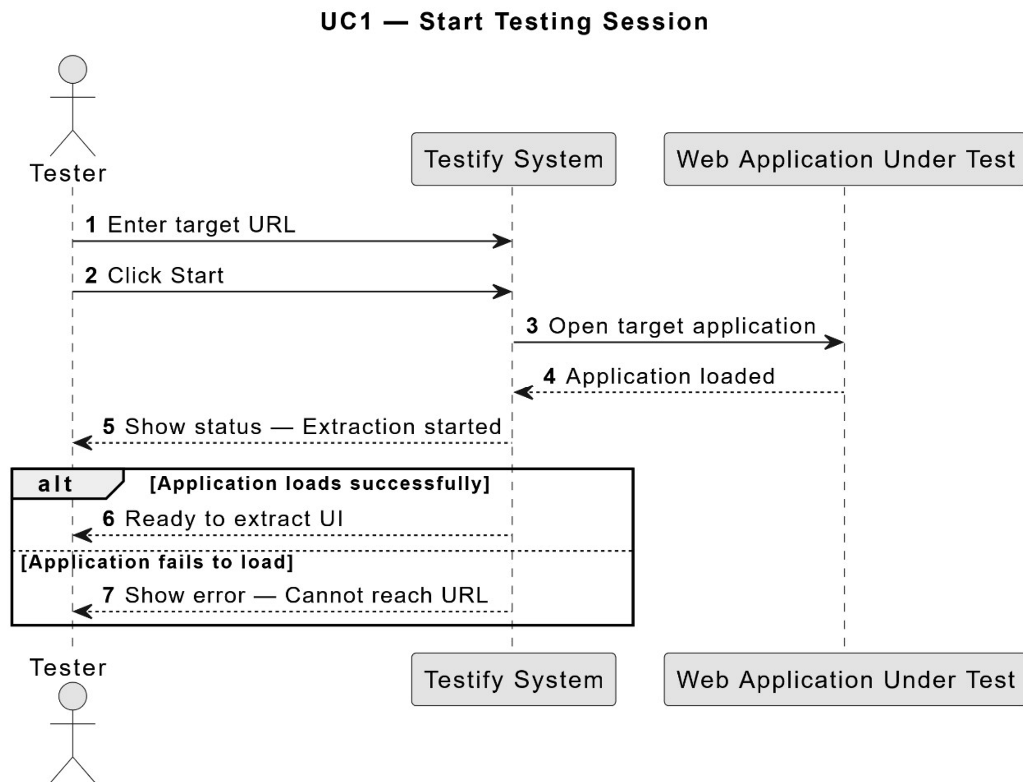

Figure 4: Start Testing Session Sequence Diagram

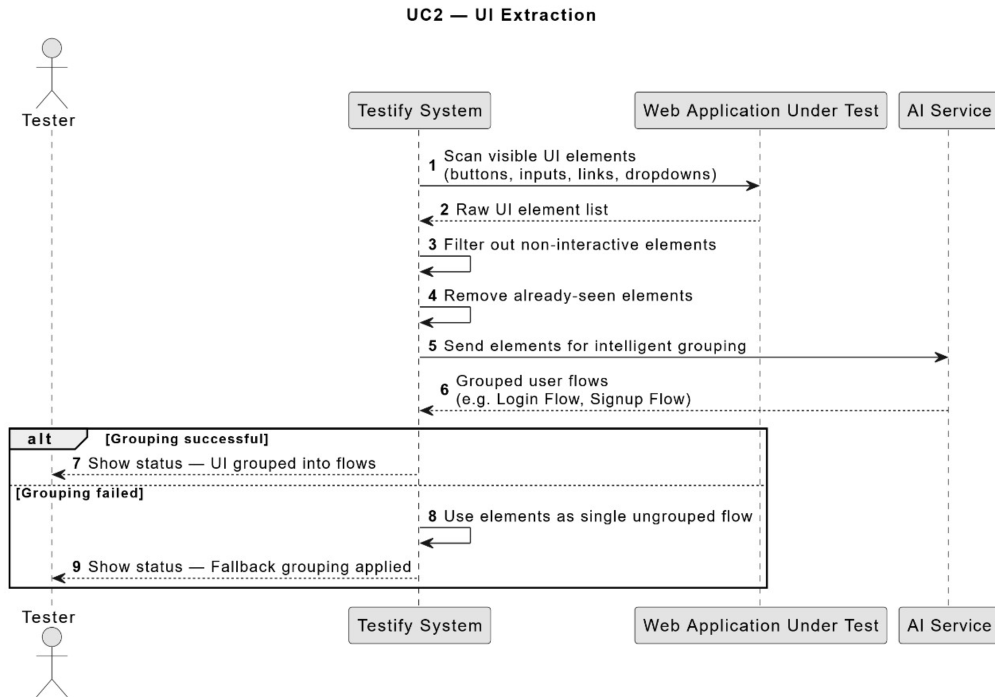

Figure 6: UI Extraction Sequence Diagram

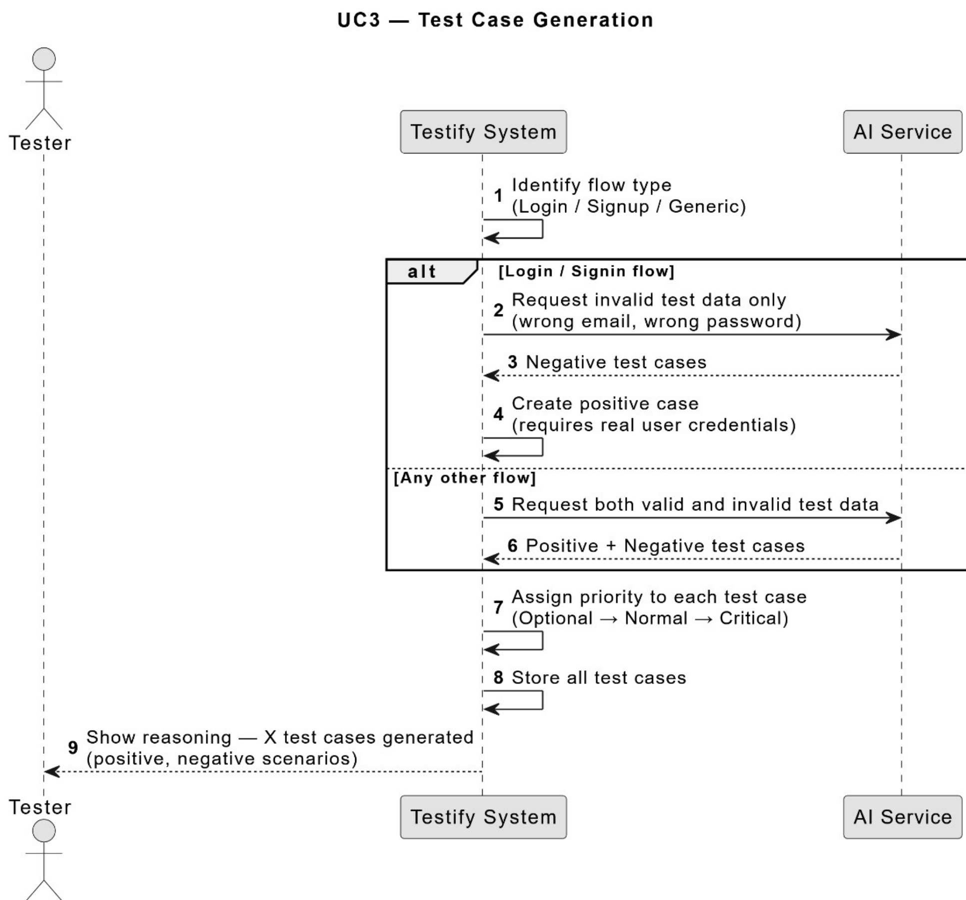

Figure 5: Test case Generation Sequence Diagram

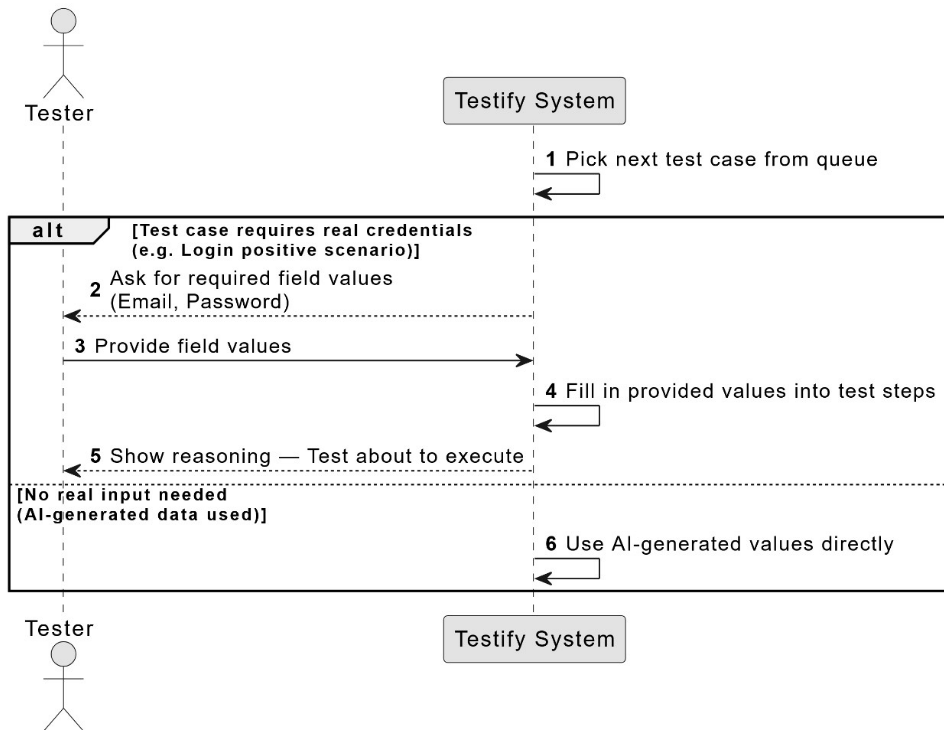

Figure 7: User Input Resolution Sequence Diagram

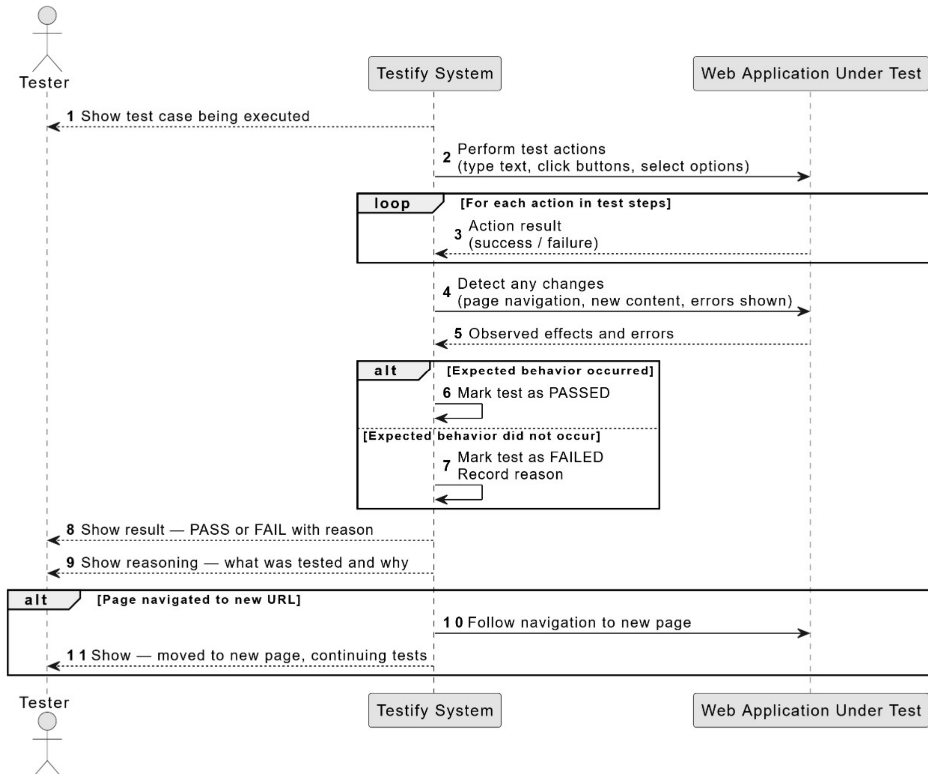

Figure 8: Test Execution & Evaluation Sequence Diagram

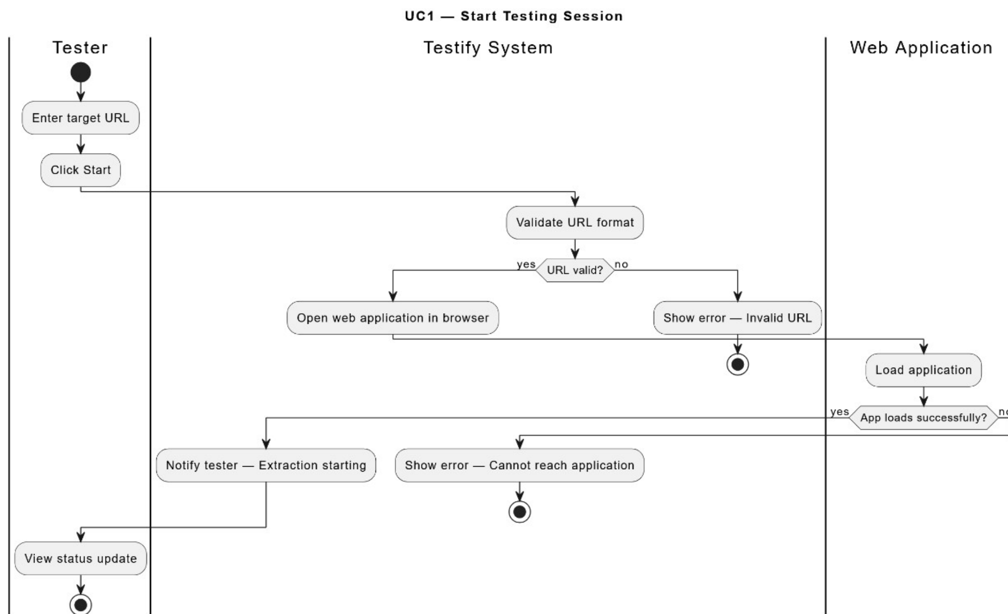

Figure 10: Start Testing Session Activity Diagram

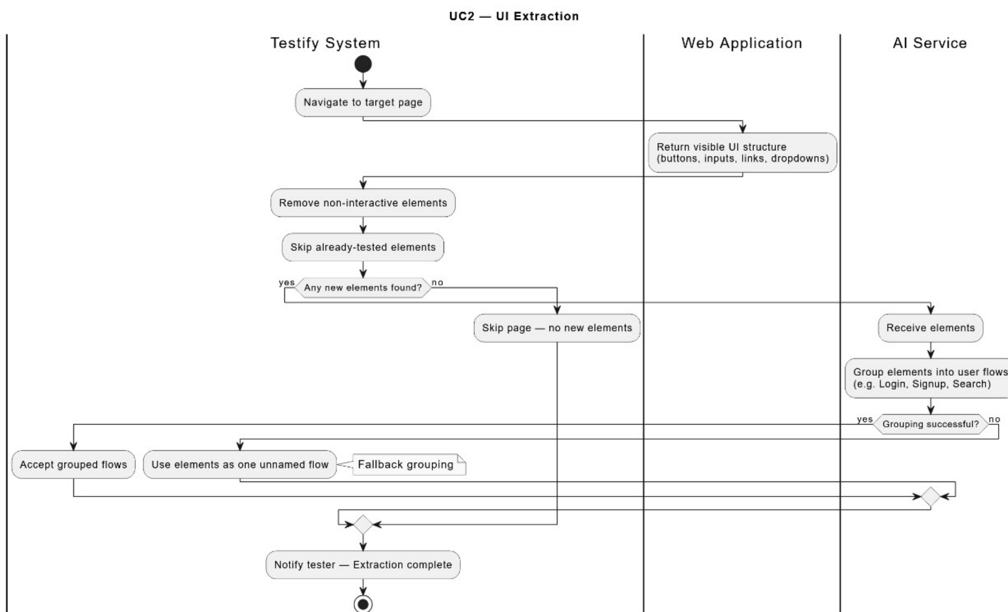

Figure 9: UI Extraction Activity Diagram

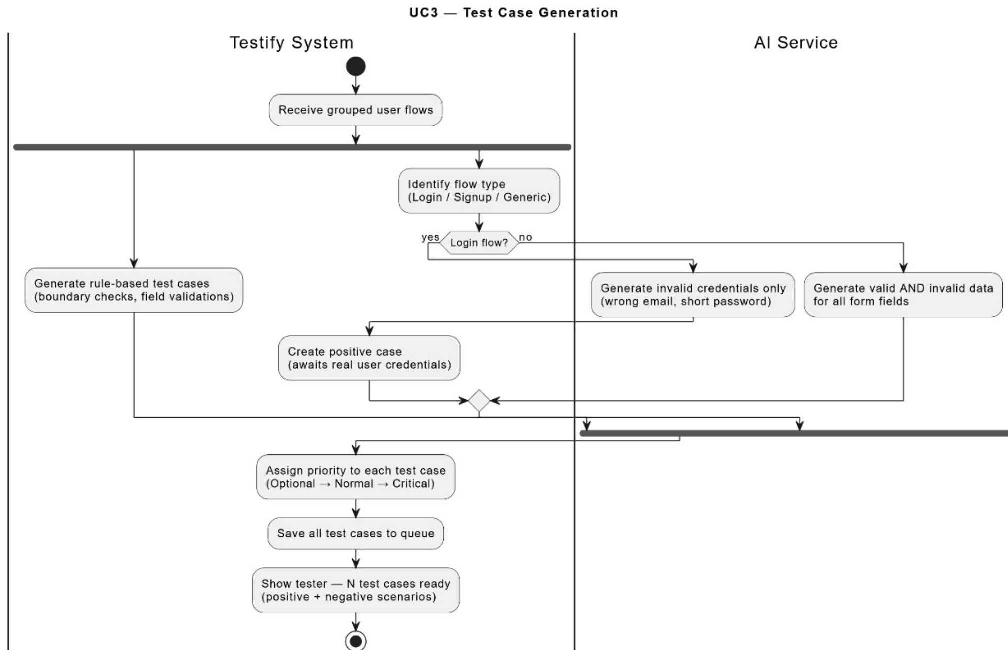

Figure 11: Testcase Generation Activity Diagram

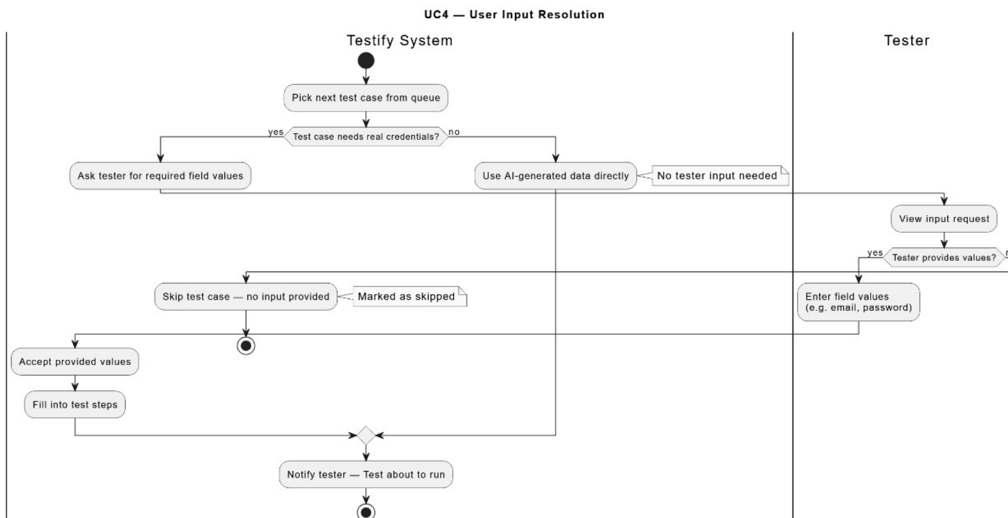

Figure 12: User Input Resolution ActivityDiagram

3.7.1 Domain Model

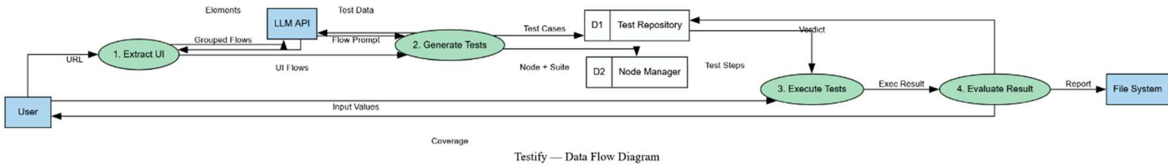

Figure 13: Domain Model of Testify

3.8 Conclusion

This chapter presented the system requirements of Testify AI. It described the main use cases, functional requirements, interface requirements, storage requirements, non-functional requirements, feasibility analysis, and analysis models of the proposed system. Together, these requirements define the expected behavior and quality of the system and provide the foundation for the design and implementation presented in the following chapter.

CHAPTER 4
SYSTEM DESIGN

System Design

The framework for Testify's architecture includes a modular approach that provides a layered grouping of modules. This module-based framework allows us to group the primary components (UI Interface, Process Manager, Browser Automation Engine, AI Logic and Reporting Tools) of the Testify application into separate blocks that can communicate with each other. In addition to providing a way to support and test individual blocks, the modularity also allows the possibility of adding enhancements or replacing a single block in the Testify application without affecting the operation of the remaining blocks.

Architecturally speaking, the Testify application uses a Linear Continuous Pipeline Model. Simply enter the URL you wish to analyze and Testify will automatically extract the layout/structure of the webpage, determine where on the webpage users may click, organize these clickable elements into logical use paths and generate test cases based upon the identified user paths. After generating the test cases, Testify will execute those test cases within the target browser and provide verification of performance. A reporting tool called "coverage map" and a reporting tool called "report", will be provided by Testify after executing the test cases. The linear aspect of this process makes it very easy to perform Smart, Simple Automation.

Testify was built from the ground up utilizing a Desktop First Model. Unlike creating a Pure Web Based Application we chose to utilize both Angular and Electron to create a desktop application that would manage all aspects of User Interface Analysis, Automated Browsing, AI Processing and Generation of Results.

4.1Design Constraints

Follows are the some design limitations impacted the construction of Testify AI:

4.1.1 Desktop-Based Deployment Constraint

We created this Angular application with Electron so it can be run from a user's own device. As such, our application architecture was developed to support one individual using the app at any time, running in isolation, on their personal device as opposed to

a multi-user model where many users access the application over multiple networks and/or in the cloud. Therefore, we made several other technology decisions with respect to how we would store and access active data locally (using a light weight storage mechanism), perform all of the application's processing tasks on the user's device and develop the overall UI layout to fit the normal monitor sizes.

4.1.2 Dependency on External Web Applications

The Functionality of Our Application Relies Directly on How Websites Are Structured & Accessible. When we developed our application, we had to be extremely cautious in relation to how frequently website developers would make changes to their website's page layout & structure; how often websites update their content dynamically as users engage with the website through user input; whether the internet connection of our application users will have sufficient bandwidth for our data to transmit back & forth successfully; and lastly what types of security measures each website developer uses that could block our ability to access the web site. Other than the previously mentioned items that can affect the functionality of our application.

4.1.3 Dependency on AI-Based Test Generation

This framework utilizes artificial intelligence (AI) assisted groupings of UI elements and creates "flow" based designs for automatic test cases. Because AI can be variable in its output, an approach was created to manage this variability during the development process. Structured Step Generation, Runtime Validation, and Fallback Logic were used to handle those variables.

4.1.4 Runtime Input Constraint

Certain web test cases will require actual end-users' data (i.e., real-time input). For example, login flows or any workflow which requires the logged on user's ID and password. Due to the ethical implications of storing user's passwords and due to security concerns, the system cannot store these sensitive items permanently. This was a major driving force behind our design process for the system; as a result we developed a runtime schema-input mechanism. The runtime schema-input mechanism will request the appropriate sensitive information (e.g., username/password) when needed by the system.

4.1.5 Lightweight Storage Constraint

We decided to use memory-based and file-based export for storing session data and reports rather than a full relational or cloud-based database. This was done because using this type of technology helped us keep the project small, easy to manage, and fast. But it means we have no method to store data for long periods of time and thus cannot measure the history of past sessions.

4.1.6 Browser Automation Constraint

Because our testing workflow uses browser automation via Playwright, along with an embedded browser/web-view interaction, we had to consider the timing constraints associated with how quickly asynchronous pages load and the random nature of changes to the Document Object Model (DOM). To increase the dependability of our system, we used waiting mechanisms, controlled sequences of executions, and validation checks while executing tests.

4.1.7 Performance and Complexity Trade-Off

In comparison to having multiple applications or services perform different functions (i.e., extraction, test generation, execution, evaluation, coverage reporting), we had to be very deliberate about balancing the quantity of features available in our application with the speed at which they performed. Due to this trade-off, we concluded that adding persistence capabilities to our application would add too many layers of complexity. Instead, we elected to implement modular processing of individual test sessions as opposed to implementing a large-scale persistent infrastructure.

4.2System Architecture

Testify AI has been designed from a modular standpoint, in that each level can operate independently but are cooperative as components of a greater whole (a single desktop application core). The system has six (6) key modules:

1. **Presentation Layer**

   The presentation layer uses Angular as its front-end framework for developing the user interface of our application. The presentation layer is composed of five (5) views which constitute the user interface of our application. These include the dashboard view, URL input view, browser panel view, chat/reasoning panel view, and coverage visualization view.

2. **Application Control Layer**

   This layer is managed by the **Electron main process**, which acts as the coordinator of the entire testing workflow. It receives user commands from the interface, starts extraction, triggers test generation, manages execution, and returns results back to the frontend.

3. **Extraction and Analysis Layer**

   The Extraction & Analysis Layer is responsible for extracting the UI structure of the target website being tested. The Extraction & Analysis Layer uses browser automation combined with accessibility-based processing to find UI elements such as buttons, links, text fields, check boxes, drop-down lists etc… Additionally, the Extraction & Analysis Layer uses grouping logic to organize these UI elements into meaningful user flows.

4. **AI-Based Test Generation Layer**

   The AI Based Test Generation Layer creates test schemas and flow-based test cases. The AI Based Test Generation Layer produces static test cases for individual UI element types. In addition, the AI Based Test Generation Layer produces dynamic test cases for workflows that may contain login/login out functionality; signup functionality; submission of forms etc…. The AI Based Test Generation Layer supports both positive and negative testing scenarios.

5. **Execution and Evaluation Layer**

   The Execution & Evaluation Layer will execute test cases produced by the AI Based Test Generation Layer within the browser. The Execution & Evaluation Layer will utilize Playwright-based interaction logic to execute test cases. After execution of test cases, the Execution & Evaluation Layer will compare actual results with expected results. If actual results match expected results then test case(s) will be classified as "pass". On the other hand, if actual results do not match expected results then test case(s) will be classified as "fail".

4.2.1 Coverage and Reporting Layer

The Coverage & Reporting Layer will record page transition events. The Coverage & Reporting Layer will display route coverage information. The Coverage & Reporting Layer will also generate test results in reportable formats like Excel.

4.2.2 System Architecture Diagram

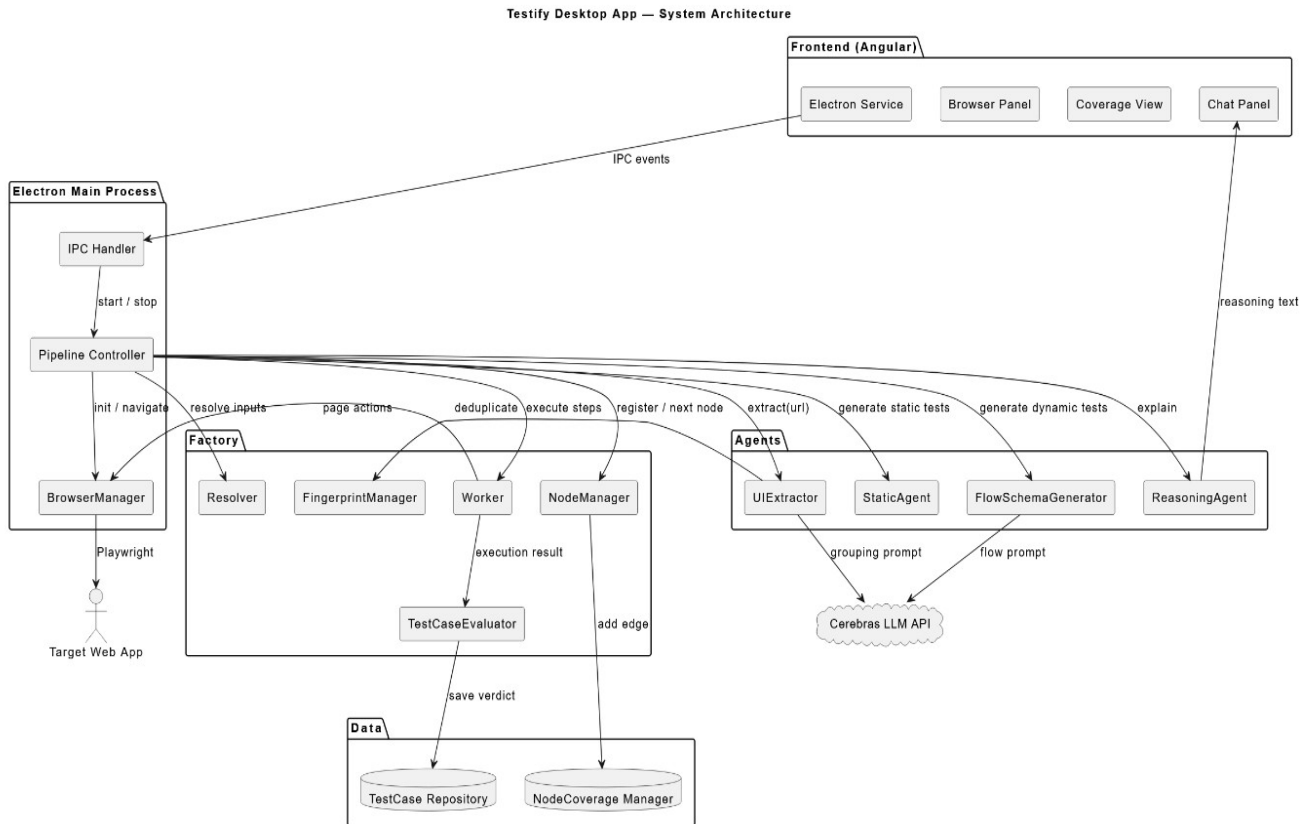

Figure 14: System Architecture of Testify

4.3 Logical Design

TestifyAI User represents a user who utilizes TestifyAI. Test Sessions are represented by all testing sessions developed for a specified user. Test Cases provide examples of specific objectives/means that are being evaluated. Each Test Step includes a Task performed on an object and contains a Target Object and Action Type. Coverage Nodes include pages viewed while testing. Coverage Edges denote the transition from one page to another based upon a Test Case. Result Reports document the pass/fail status of a Test Case along with the Reason(s) for passing/failing and Summary Data to be exported.

From a functional perspective, TestifyAI will work in two ways logistically: First, Users will begin their use of TestifyAI with one Test Session at a time. Second, Users can develop multiple Test Cases within any individual test session. Third, each Test Case will consist of multiple Test Steps. Fourth, after completion of all Test Cases

within a test session, they will contribute to both Page Coverage and Reporting Outputs. Fifth, The logical organization of Testify AI allows for effective means to organize executions and interpret results.

4.3.1 Domain Model Diagram:

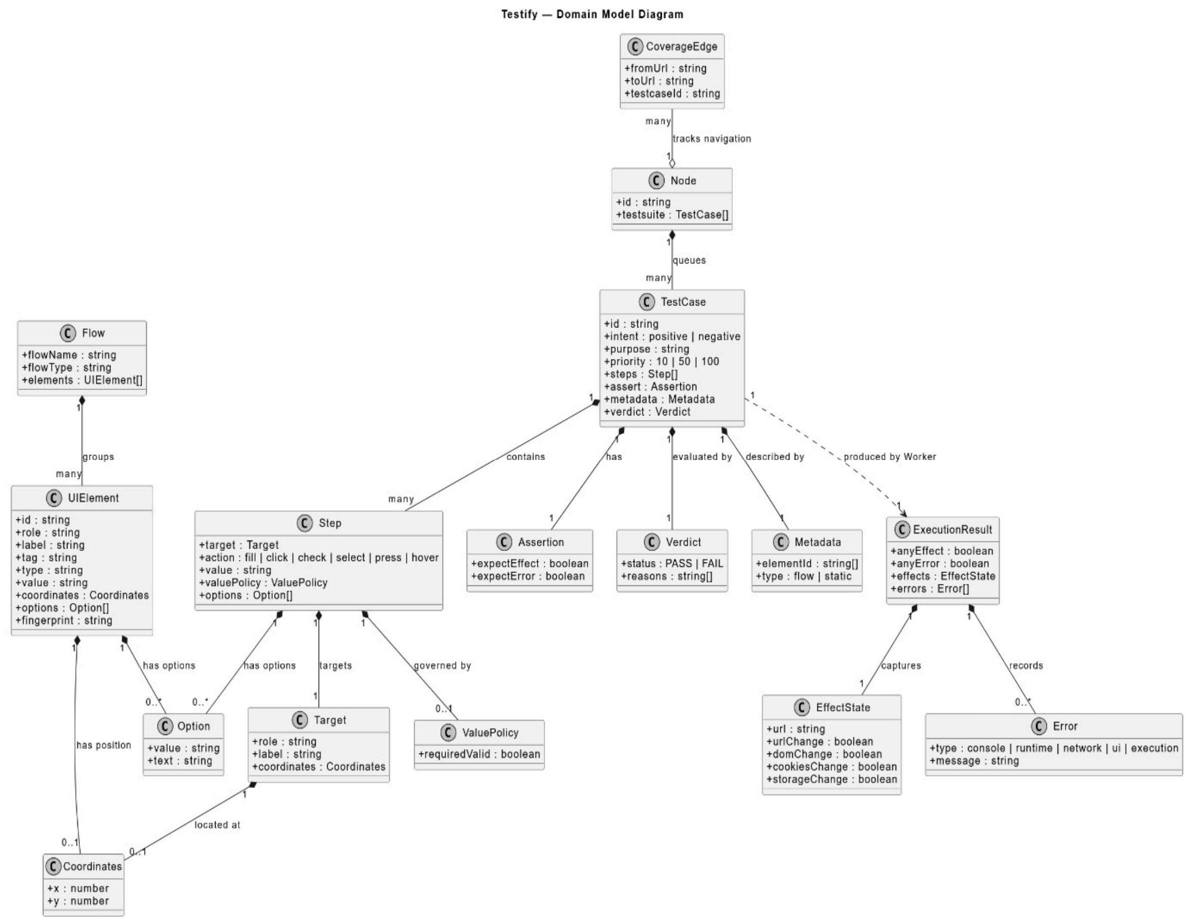

Figure 15: Domian Model Diagram

4.4 Dynamic View

The Dynamic View allows you to see how the application works with different components of the application when you run the application. When a user types a URL into the Search Bar located in the Frontend of the Application, the Frontend sends the URL entered by the user to be processed by the Electron Controller.

Upon receipt of the request made by the user, the Electron Controller launches your web browser (regardless if it is launched on your Desktop or Laptop Computer), and begins to extract all of the UI Objects (buttons, links etc.) that make up the overall

structure of the User Interface (UI). Once all of the UI Elements have been extracted, both the Static Test Generation Module and Dynamic Test Generation Modules receive copies of the elements. Each module creates temporary test cases based upon what they found in their extraction process. Each test case created by both modules is executed individually. Prior to executing any of the test cases, the System determines whether any of the individual steps in each test case require valid user input (valid login credentials for example). If any valid user input is required prior to execution of any step(s), a request is made back to the UI for those specific values. After the user has filled in those requested values, the Execution Module executes the actual test actions inside the Automated Web Browser.

Following execution of the test actions, an Evaluator compares what was observed in the browser versus what was expected (or an Error Condition) and lastly, the System updates coverage information regarding what was executed and presents this back to the User in the form of a display in the interface. Following completion of a session, results from testing sessions are exported.

4.4.1 Activity and Sequence Diagrams

Figure 16: Main Testing Pipeline Sequence Diagram

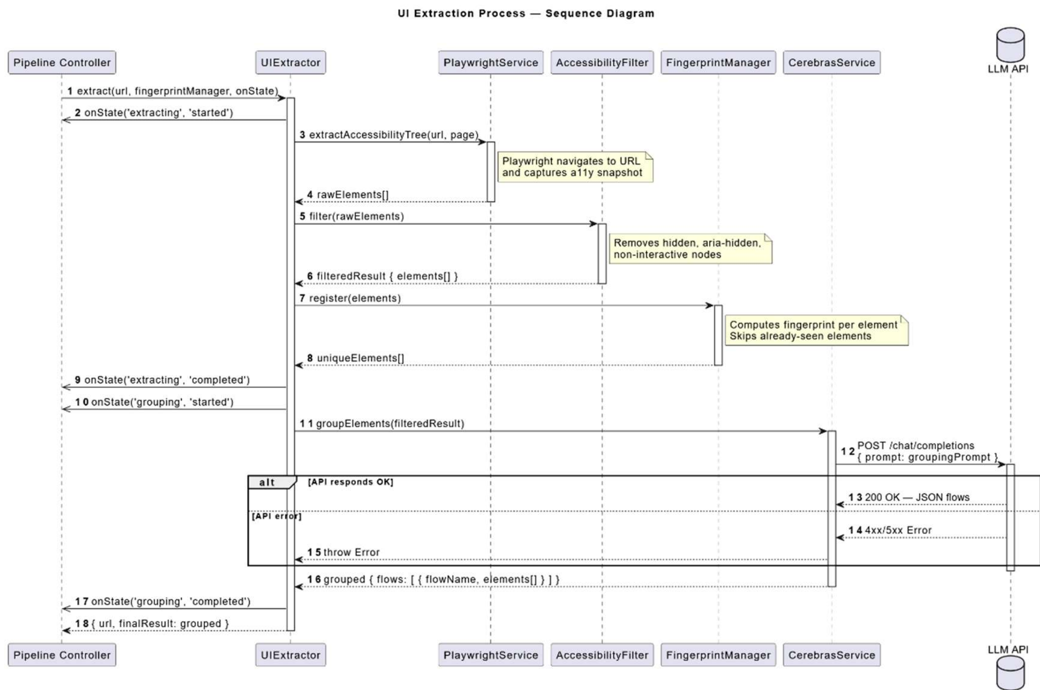

Figure 17: Execution Process Sequence Diagram

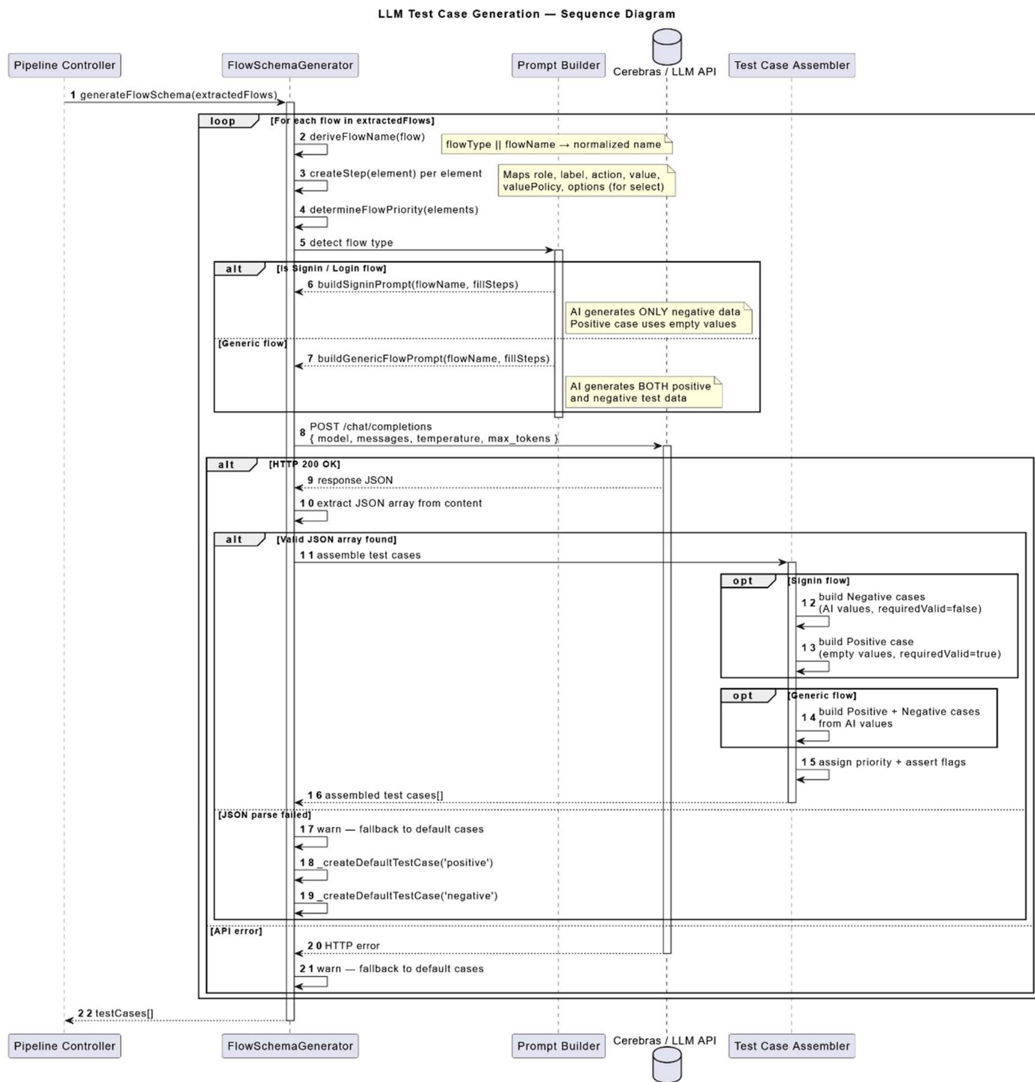

Figure 18: LLM Testcase Generation Sequence Diagram

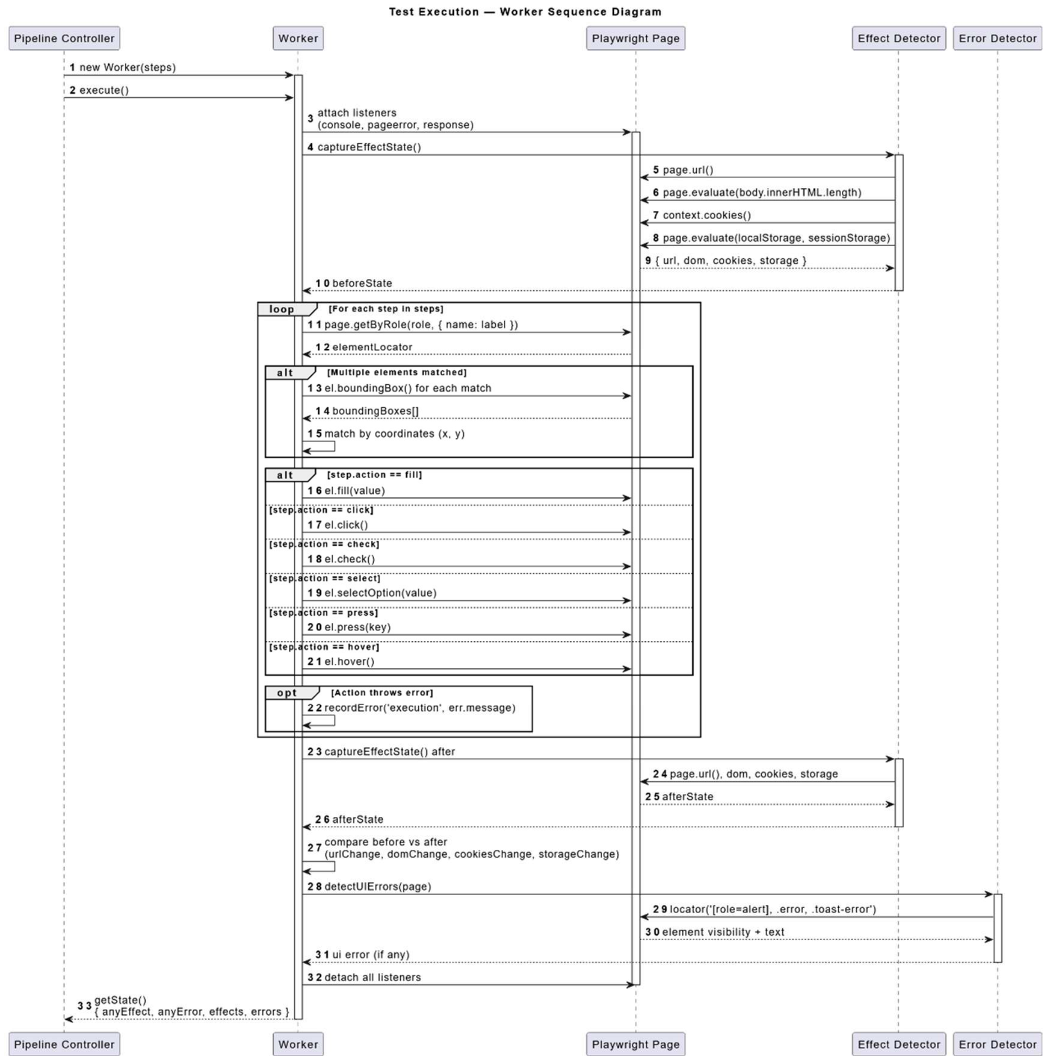

Figure 19: Test Execution Sequence Diagram

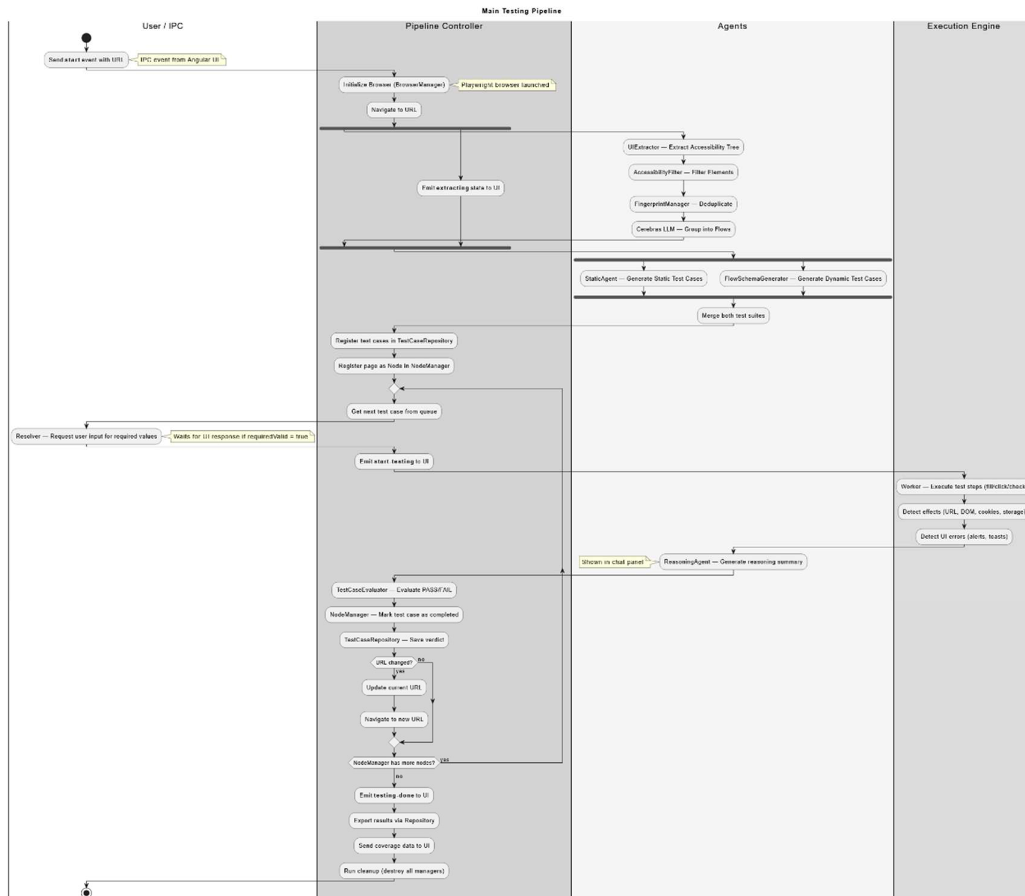

Figure 20: Main Testing Activity diagram

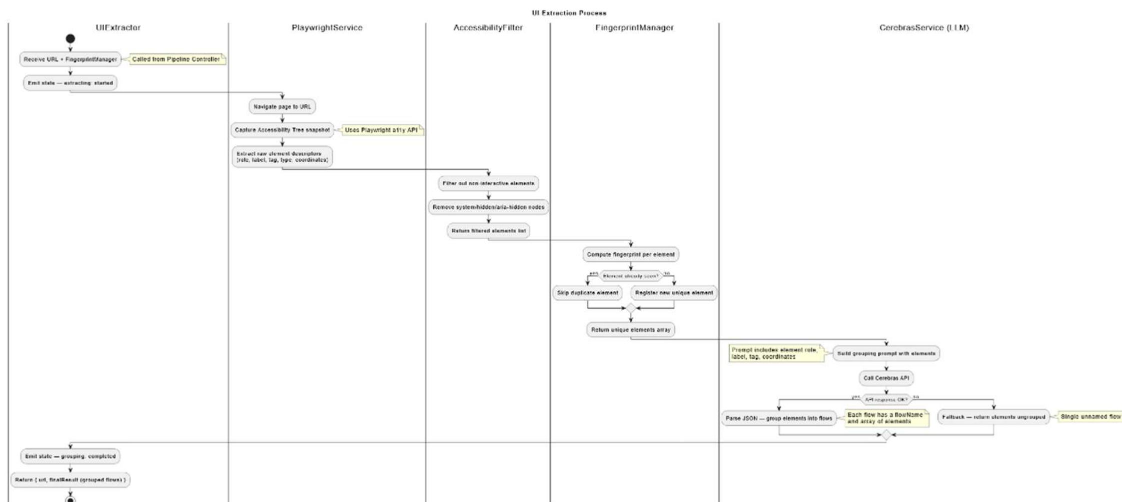

Figure 21: UI Extraction Sequence diagram

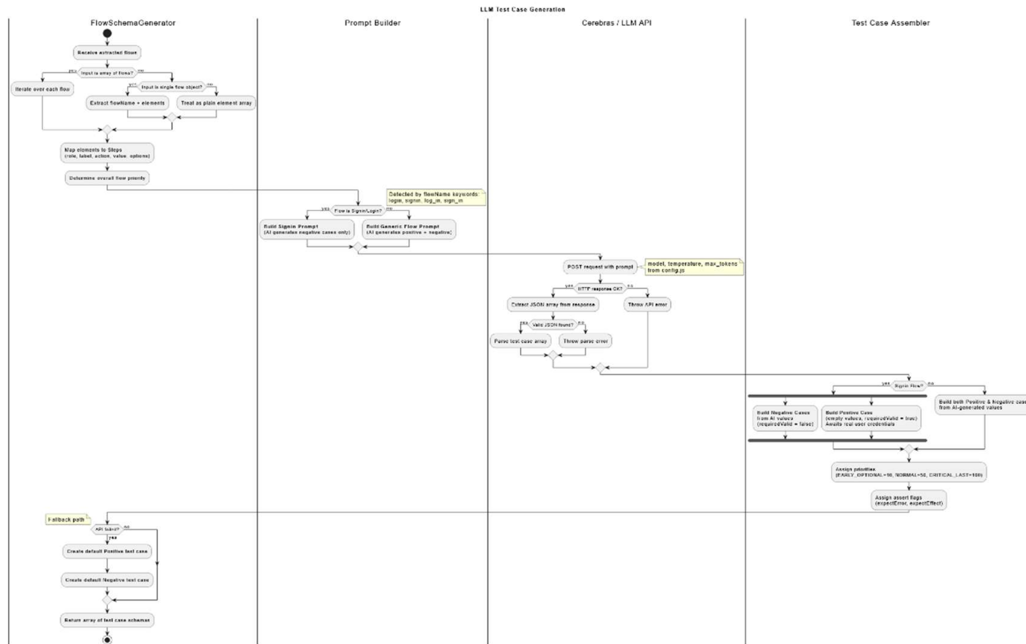

Figure 22: LLM Test Case Generation Sequence diagram

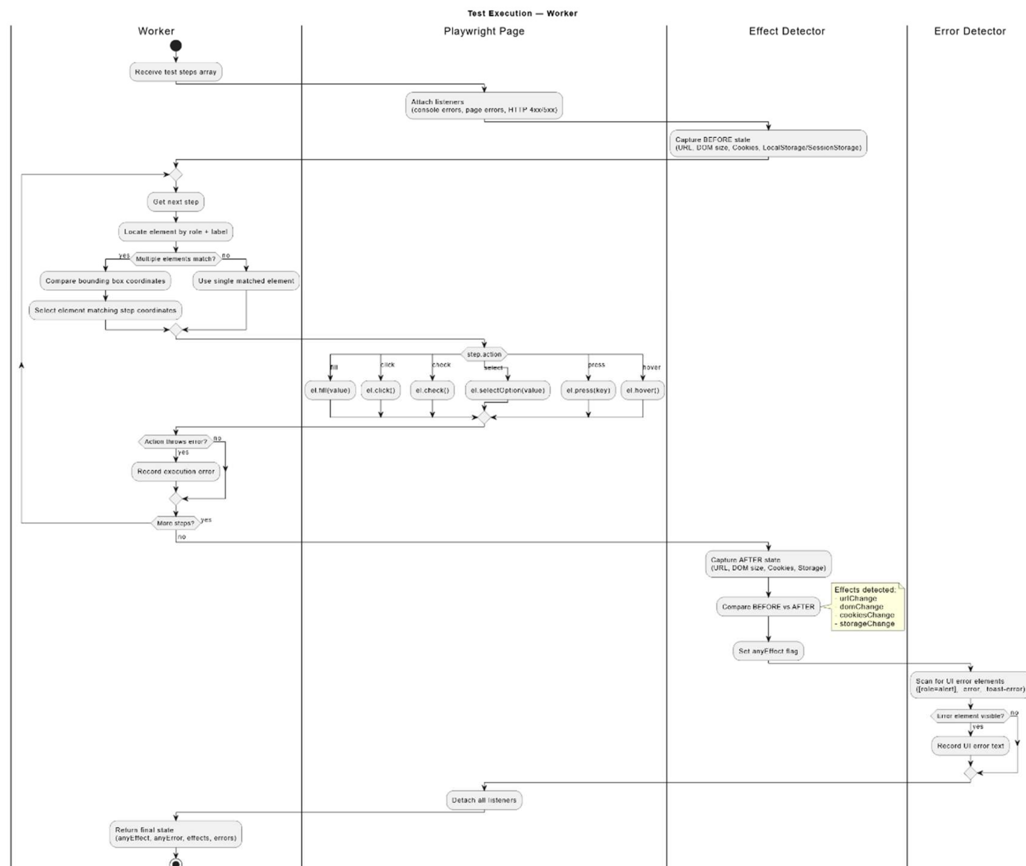

Figure 23: Test Execution Sequence diagram

4.5 Component Design

1. **Frontend UI Component**

   Angular pages and reusable parts of the user interface are contained and displayed in the front-end UI component. The front-end UI component determines how the layout of the dashboard, testing screen, browser viewing area, chat/reasoning panel, and coverage graph view should appear.

2. **Electron Controller Component**

   The Electron control component manages the lifecycle of the application. The Electron control component acts as a bridge that enables communication between front-end and back-end logic. Events are processed through inter-process communication (IPC), and the majority of application operations are orchestrated by the Electron control component.

3. **Extraction Component**

   Each page's structure is identified and extracted by the extraction component. Each page's unnecessary elements are identified and removed by the extraction component. All extracted data is compiled into groups based on type of element; therefore, it may be used to generate tests.

4. **Static Test Generator Component**

   Static test cases are generated from identified UI elements by the static test case generation component. Primarily non-interactive items such as buttons and links will be focused upon by the static test case generator.

5. **Dynamic Flow Generator Component**

   Flow-based test cases using artificial intelligence are created by the dynamic test case generation component. Interactive scenarios for users were designed to be identified by the dynamic test case generator.

6. **Execution Component**

   Browser actions are executed in either the web-view or the browser automation environment through the Execution component. Clicking, filling, selecting, toggling, navigating are examples of actions that may be performed.

7. **Evaluation Component**

   An expected result is compared with a result occurring when executing a test case by the evaluation component. Successfulness or failure of the executed test case is determined by the evaluation component.

8. **Repository and Coverage Component**

   All existing test cases created while they exist are maintained in a record form by The Repository and Coverage Component. The Repository and Coverage Component also provides a foundation for building a visual representation of all pages that have been viewed and transitions made between those views.

9. **Export Component**

   A final report file including all created test cases along with their results is produced by the export component

4.5.1 Component Diagram

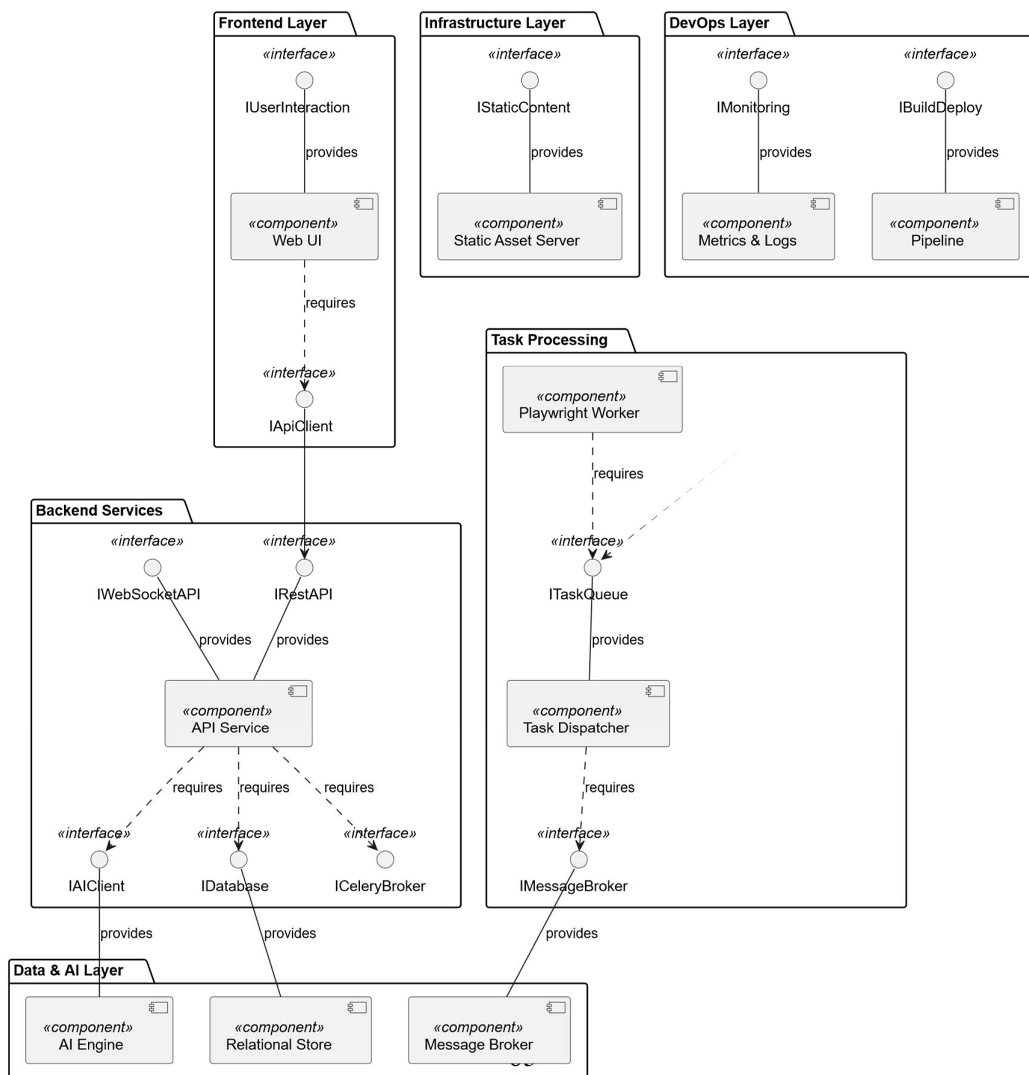

Figure 24: Component Diagram

4.5.2 **Package Diagram**

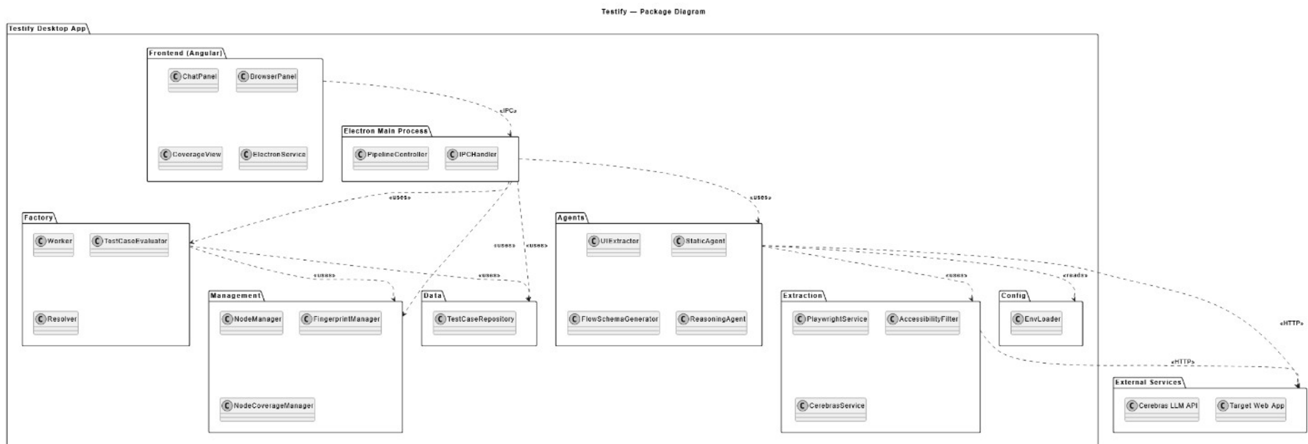

Figure 25: Package Diagram

4.6 Data Models

Testify AI uses a relatively minimalist session-based data structure. The basic components are listed below:

- **TestCase**
  Contains test case id, purpose of test, what you want the test to do, level of priority, values you assert in your test case, meta-data on the test case, and status of where the test case is currently at during execution
- **Step**
  Includes Target Role, Target Label, X,Y coordinates, Action Type, input value, and Value Policy.
- **Assertion**
  Includes Expected Effect and Possible Error Conditions.
- **CoverageNode**
  ncludes page identifier (as an object), page URL, and navigation edges which point to other coverage nodes.
- **CoverageEdge**
  Includes source page, destination page, and test case ID associated with this edge.

- **ExportRecord**
  Includes summarized result fields such as purpose of test, priority, intent, assertion summary, and status.

Follow model represents how testing information is organized during Execution.

4.6.1 Deployment Diagram

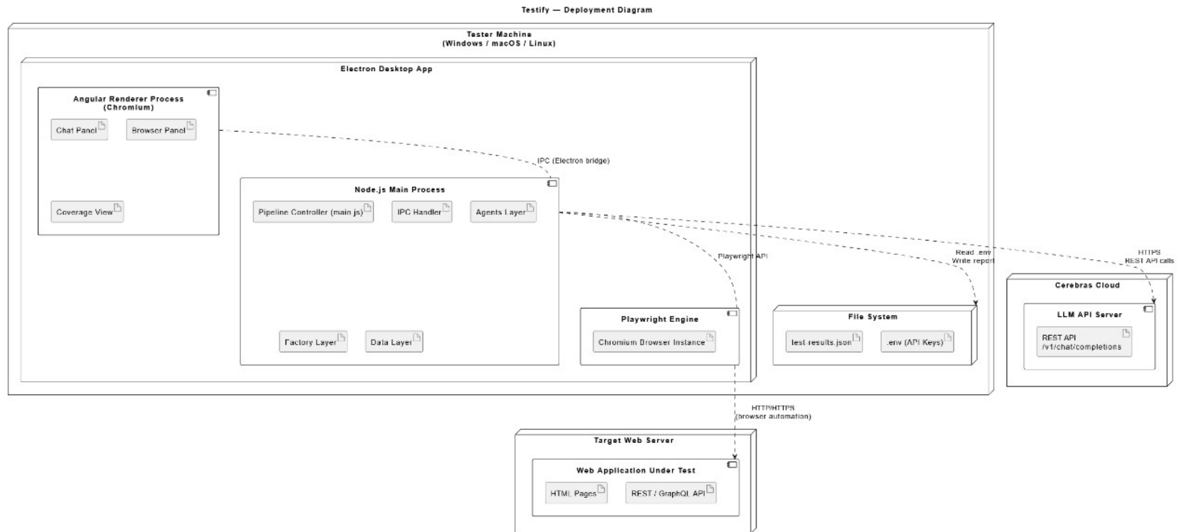

Figure 26: Deployement Diagram

4.7 User Interface Design

Testify's User Interface (UI) has been created to enable simple interaction during the testing process. Testify's UI is easy-to-use, visually indicates progress through each step of the test and gives complete control to the user throughout the entire testing process.

The first part of the UI presented to the user is called the dashboard. This screen is used to allow the user to choose which module they wish to run. Once the system testing module has been chosen, the main test interface screen appears. When opened, there is a centered area called "Input Area" and it contains a single input field. The user enters the URL of the desired website into the input field and once entered clicks submit. The input field is removed from the middle of the screen and divides vertically. On one side of the vertical divide is the "Browser Panel." Inside the browser panel is the URL that was entered. All user-testable interactions are

performed inside this panel. On the other side of the vertical divide is the "Chat/Reasoning Panel." Testify uses this area of the screen to communicate several types of messages to the user. These include Progress Messages; Extraction Summaries; Test Plans; Result Feedback; and Credential Requests. Once execution of a test plan is completed, the User can navigate to the Coverage View Screen. The Coverage View Screen is a graphical representation of all page transitions that were tested.

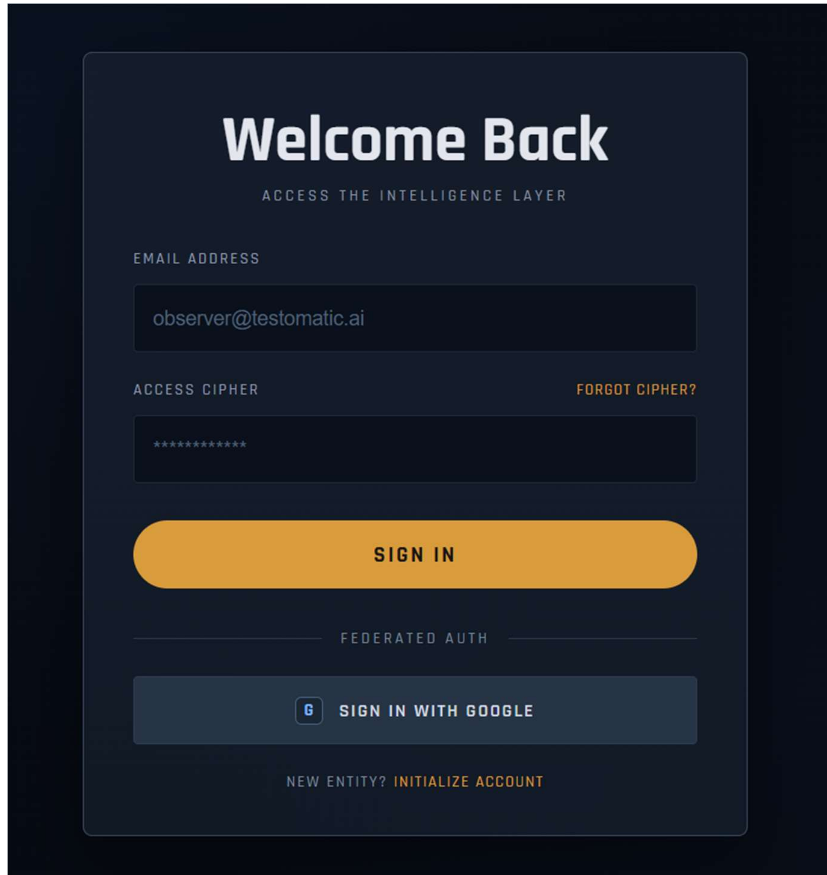

Figure 27: Login page of Testify

Figure 28: Dashboard Page

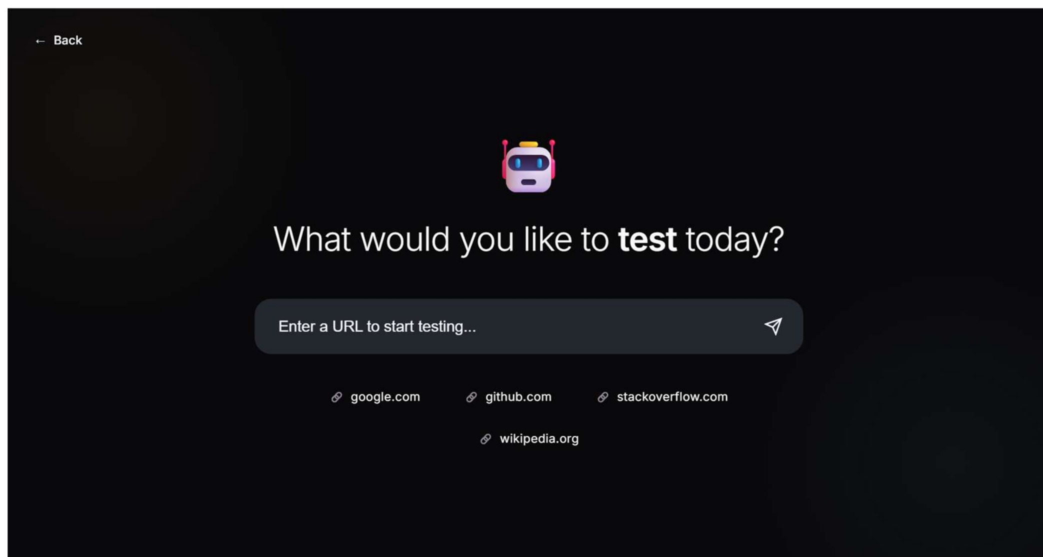

Figure 29: Search Engine Page

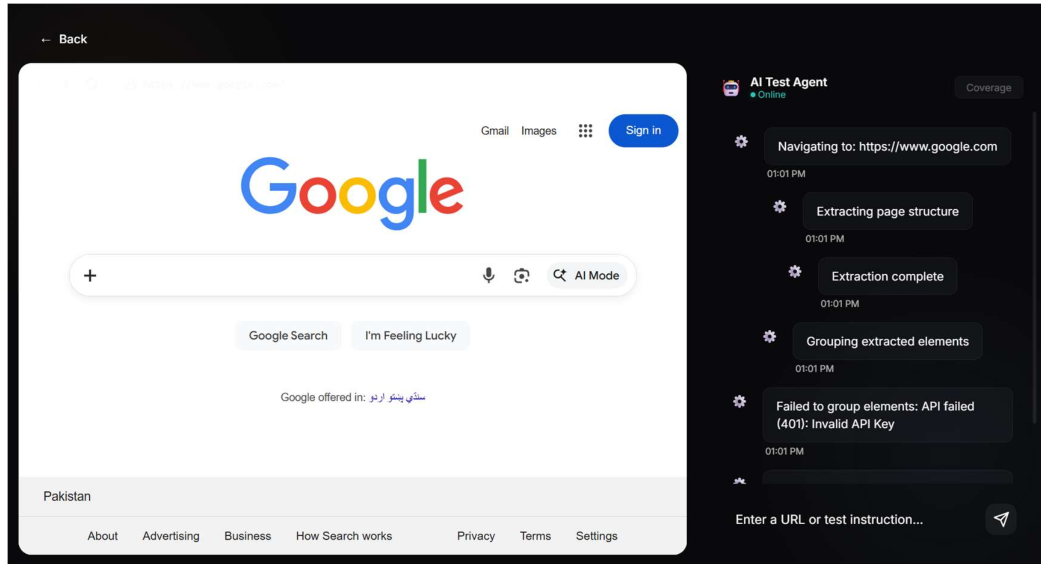

Figure 30: Execution page

4.8 System Prototype

The Testify AI system evolved from a low-end desktop application based upon a workflow design concept into a high-end desktop application. Initially, we were focused on developing three primary features to include in the system: identifying page layouts, automating tests and giving end-users instant visual feedback as to which tests are executing. We developed a first pass at these individual steps using the simplest forms of workflows and wireframes.

4.9Conclusion

Our current prototype has now been completed and is a fully functional desktop application (Angular & Electron), which will enable end-users to input a URL live; load pages; extract elements from the DOM; generate tests by using an artificial intelligence component; manage login information; track execution of tests live on-screen; display output generated by AI; provide metrics related to which portion of their application is being tested and export reports. As opposed to a mock-up intended for use during testing only, this prototype demonstrates an operational example of each component of the architecture we have designed.

CHAPTER 5
SYSTEM IMPLEMENTATION

System Implementation

5.1 Introduction

This section presents the implementation of Testify AI, an intelligent desktop-based web testing system designed in this research project. This stage involved transforming the proposed system architecture into a fully functional application. It will utilize its capability to analyze web pages, generate test cases, automate interactions against those pages, evaluate the outcome of those tests and display how much of that particular page was covered during those tests. Testify AI is being built utilizing a combination of the latest desktop and web based technologies, focusing heavily upon modularity, automation, and use of AI within the testing process.

5.2 Development Environment and Technologies

Testify AI's implementation utilized the following key technologies to create the core functionality of the system:

- Angular was chosen as the primary technology for building the user interface (front-end) due to its component-oriented nature and structured approach.
- Electron was chosen as the packaging method for converting the front-end into a desktop application, allowing users to have access to native desktop features such as file dialogues and windows along with providing local process management capabilities.
- Playwright was also chosen as it offers the most up-to-date browser automation options available today and is able to provide a high degree of reliability when interacting with dynamic web applications.
- TypeScript was chosen as the main language for implementing all of the logic required by the user interface (front-end).
- Node.js was chosen for executing server-side processes, coordinating the various automation modules, and facilitating communication between the different components of the system.
- Mermaid was chosen for visualizing coverage graphs.
- XLSX library was chosen for exporting test result data in Excel format.

5.3 Overall Implementation Strategy

This system was developed as a series of independent, modular elements. All major architectural elements were developed independently. As mentioned earlier, the build-out of this system occurred within six general categories or areas:

1. Ui Implementation
2. Desktop application control & IPC communication
3. Extraction & analysis of web pages
4. Generation of test cases (static & dynamic)
5. Test execution & evaluation
6. Reporting Coverage data & visualization

5.4 Frontend Implementation

The frontend of Testify AI was developed using angular. It consists of multiple views, as well as reusable components. A robust environment was established in which to create tests for applications.

5.4.1 Home Interface

Users were presented with the home view when initially launching the application. The home view consisted of card representations of available testing modules and contained links to navigate to the system testing view. In order to present the best possible user experience, the home view was designed to be very clean, simple, and easy to read.

5.4.2 System Testing Interface

A dedicated angular web-view was created for the system testing view. When launched, this view presents a single entry field to accept the URL the user desires to submit for testing. Upon submission of the URL, the layout of the view transforms to display two distinctive panels.

- a browser view panel appears on the left side of the screen.
- a chat/reasoning panel appears on the right side of the screen.

Each of these panels allows the user to observe how the browser communicates with their website and review how the system develops its conclusions via its reasoning abilities.

5.4.3 Chat and Interaction Panel

In order to provide means for interacting with the user regarding various aspects of their testing experience such as extraction progress, test plan, execution status and results of test results; a chat panel was incorporated. The chat panel was also used for user input during runtime as determined by the system and deemed necessary for runtime fields (e.g. Username/password). Interactions with this panel utilized message models and event emitters so that the frontend could dynamically react to requests sent by the Electron process.

5.4.4 Coverage Visualization Interface

An additional component was added to provide means for displaying graphically which routes/pages had been tested according to Mermaid generated charts. This component also allowed users to interact with test case detail information allowing them to analyze relationships between page transition and executed test cases.

5.5 Electron-Based Desktop Integration

Electron was selected as the platform to bridge the gap between the angular frontend and the backend processing logic that supported the system level automation. The Electron main process acts as an application-wide controller.

The Electron main process is responsible for handling several key functions:

- creating the application's main window
- utilizing Electron features to load and run the angular frontend
- managing communications between processes using IPC (interprocess communications)
- coordinating actions related to extraction, test generation, execution and exporting
- sending relevant data and/or feedback regarding progress/status/results back to the user interface

As previously stated, Electron utilizes IPC events like start, stop, request-schema, start-testing, testing-done and Coverage-data to provide communication between the

frontend and backend of the application. Since these events are event-driven, they permitted the application to maintain activity/response while performing background processing tasks.

5.6 Web Page Extraction Implementation

Our extraction module extracts and pulls useful interactive components from target websites. To accomplish this task, we used a combination of accessibility parsing and browser automation to obtain interface information. Below are the steps taken to extract the desired interactive components:

Step 1: A user inputs the URL he/she wants to use.
Step 2: The electron controller opens the browser by initiating the automation process.
Step 3: The Extraction Service is opened to display the targeted webpage.
Step 4: All interactive components on the displayed page are gathered.
Step 5: Any unnecessary or non-related components that were gathered in Step 4 are filtered out.
Step 6: Any duplicate components that were extracted in Steps 4 and 5 are fingerprinted.
Step 7: The layout of all remaining components is organized in a way that makes sense to end-users.

Our approach differs from simply extracting the raw HTML code of a website, because it focuses on the aspects of a website that really matter to its users. For example, instead of recognizing "buttons", "links" etc., we recognize "clickable buttons", "hyperlinks", "input fields", "checkboxes", and "drop-downs".

5.7 Test Case Generation Implementation

We have implemented two different ways to create test cases. These include creating static tests and implementing flow-based dynamic tests.

5.7.1 Static Test Generation

Static Test Generation creates simple tests for individual buttons/links etc. to verify their functionality (i.e. whether they perform their intended function). This is done by applying a number of predefined rules to determine if there is a valid stand-alone item that exists. If the rules indicate that there is a valid stand-alone item, then a test script is generated that reflects that determination.

5.7.2 Dynamic Flow-Based Test Generation

Dynamic Test Generation is designed to handle complicated user interactions (e.g. registering/login, filling out forms, browsing through several web pages) and breaking these down into step-by-step scripts that can be executed. Not only does Dynamic Test Generation produce successful paths, but also error paths and other possible actions that include:

- Fill
- Click
- Select
- Check
- Un-Check
- Toggle
- Hover
- Navigate

We have integrated artificial intelligence logic to generate realistic test data so that we can simulate testing scenarios that would apply to most users. When a flow encounters a log-in barrier, the system will pause and ask for real-world credentials rather than guessing them.

5.8 Runtime Input Handling

Another important component of our product is processing input during live tests. Many test-steps require specific data that standard placeholder values cannot provide especially for authenticating user-flows.

To solve this problem we developed a schema-request workflow:

1. The execution controller determines which test-steps need valid user input.
2.The controller signals to the front-end.
3. The front-end shows a simple input field in the chat-panel.
4. The user enters the required information via typing.
5. The system adds those values to the pending test-steps.
6. The test continues without interruption.

By solving this problem, we allow our tool to work on live sites and, since no data is stored after the current session, we keep sensitive data encrypted in memory until completion.

5.9 Test Execution Implementation

Test-cases are executed using Playwright for browser automation and injecting JavaScript directly into the page. After loading a test-case, the system performs each test-steps in sequence.

Implementation includes support for multiple interaction-types:
1. Clicking elements
2. Typing text into fields
3. Selecting options from dropdown lists
4. Toggling switches
5. Checking/unchecking boxes
6. Browsing hyper-links
7. Pressing keys on the keyboard

We have also provided a browser-panel allowing you to manually manipulate the web-view and execute your own actions, while visualizing how those actions occur. Each test-steps are converted into direct actions against the browser. Elements are identified via role, label-text, and previous screen position matches.

Besides providing increased transparency as to what elements are being interacted with, each element being interacted with is highlighted in red while being manipulated, so you always know exactly what area of the page is currently being tested.

5.10 Test Evaluation Implementation

Once the test is finished, Testify determines if there are any discrepancies in what happened compared to what should happen. Testify determined that there would be two types of evaluations to measure the performance of the test.

* Did the system work correctly?
* Was the correct error message shown?

Testify compares the actual results to the expected results to grade the test. If the actual results matched the expected results, Testify gives a PASS. If the actual results did not match the expected results, Testify gives a FAIL. In addition to giving a

pass/fail rating, Testify also describes why the pass/fail determination was made. Therefore, Testify does not just hit buttons; it verifies that the results mean something.

5.11 Repository and Coverage Implementation

While Testify runs through the tests, it uses a small footprint, fast-access memory storage method to handle everything related to the tests.

5.11.1 Test Case Repository

Testify built a live database function to store generated test cases, identification numbers for each test case, any details about each test case, the test case score for each test case executed, etc. Because Testify is able to pull up all of the data that was saved during the testing and import it into a file for exportation purposes, Testify's repository allows users to get their hands on all of the data they need once testing has been completed.

5.11.2 Coverage Manager

In order to give users a better understanding of where on the application were being evaluated during testing, Testify created a coverage manager. The coverage manager tracks every page on a website that was opened during testing, along with every link clicked on during testing. Each page opened is considered to be a "stop" on a map, and each link clicked is considered to be a "link" from stop to stop on the map. Once testing is completed, Testify produces a visual depiction of all locations visited using the data collected by the coverage manager. This will allow users to look at how well the application was tested, rather than just looking at lists of passes/fails.

5.12 Export and Reporting Implementation

Since creating reports quickly and easily for users is important to Testify, it added an export option that captures all test results and exports them directly into an Excel spreadsheet. Each spreadsheet produced includes the following main points for each test:

• Priority level of the test

• Reason for conducting the test
• Intended purpose of conducting the test
• Summarization of all assertions that were made for each test
• Results after completion of the test

Testify used the XLSX library to create this feature so that users will have options when importing the exported data into their project documentation or further analysis.

5.13 Integration of System Modules

Since creating reports quickly and easily for users is important to Testify, it added an export option that captures all test results and exports them directly into an Excel spreadsheet. Each spreadsheet produced includes the following main points for each test:
• Priority level of the test
• Reason for conducting the test
• Intended purpose of conducting the test
• Summarization of all assertions that were made for each test
• Results after completion of the test

Testify used the XLSX library to create this feature so that users will have options when importing the exported data into their project documentation or further analysis.

5.14 Implementation Challenges

There were many obstacles that arose while developing Testify. The biggest obstacle faced was that websites are unique in construction; therefore, extracting pure data from webpages can be difficult. The second biggest obstacle faced was that dynamic sites can change while you are evaluating them; therefore, making it difficult to identify specific buttons/hyperlinks in test cases. The third major obstacle faced was that AI-generated test cases can be random; therefore, requiring a manual review prior to processing to assure consistency. The fourth big obstacle faced was that certain tests require user input (i.e., password input); however, password input cannot stay in memory for extended periods of time; therefore, creative solutions needed to be found to address this issue.

5.15 Conclusion

This Chapter Demonstrated How to Develop Testify AI. In this Chapter, we showed how Testify AI was developed using the Angular framework for Web application development, an Electron Framework for Desktop Applications, the Playwright library for generating test cases for web applications and other lightweight modules for data handling, as well as AI-assisted generation modules. We provided the Implementation Details about how the Design Concepts were turned into a working Desktop Application which could Identify UI Elements; Generate Test Cases; Run Tests; Evaluate Results; Display Coverage Results; Export Reports. The next Chapter will be discussing how we tested our results, evaluated them and discussed our implemented system.

CHAPTER 6
SYSTEM TESTING AND EVALUATION

System Testing & Evaluation

6.1Test Strategy

Testing approach to ensure each layer is working independently from others before they are combined for full application validation. Each layer is being evaluated individually before they are combined in order to validate that each layer is working individually and collectively as one application.

Layers include:

COMPONENT Testing - to evaluate performance of individual components.

Unit Testing - to evaluate the function of individual modules or smaller groupings of code (functions).

Integration Testing - to evaluate how different components interact.

System Testing - to evaluate the entire application as one unit.

Functional test Cases - to evaluate testify's ability to fulfill actual world usage scenarios as expected.

Ultimately, we want to demonstrate that this application will "do what it is supposed to" do. This includes the following:

- Input a URL;
- Identify the proper interactive elements on the webpage;
- Create reasonable test Cases based upon the identified items;
- Request user input at the appropriate time;
- Effectively perform browser actions (clicking & typing);
- Determine the resulting outcomes;
- Generate a readable report summarizing what was tested; and,
- Export/Save data without failure.

6.2 Component Testing

Before combining all components together through integration Testing, we completed COMPONENT-level Testing to validate that our key building blocks of software are working properly.

6.2.1 Frontend Interface Component

We applied pressure on our angular front-end to see if it would allow for quick/accurate loading of the dashboard, URL entry field, split screen browser/chat views, and coverage maps and for immediate response to user interactions. Specifically, we reviewed menu navigation behavior, whether buttons were performing their intended action, if messages were displaying properly and how the application reacted to changing visuals while running.

6.2.2 Electron Communication Component

We applied pressure on our angular front-end to see if it would allow for quick/accurate loading of the dashboard, URL entry field, split screen browser/chat views, and coverage maps and for immediate response to user interactions. Specifically, we reviewed menu navigation behavior, whether buttons were performing their intended action, if messages were displaying properly and how the application reacted to changing visuals while running.

6.2.3 Extraction Component

We tested the web data EXTRACTION utility to verify that it could properly scan a website's HTML structure, identify the only relevant interactive elements (buttons & links), filter out unwanted junk code, and deliver clean and organized data packages to the ai for creating tests.

6.2.4 Test Generation Component

We validated both static and dynamic ai test generators to ensure they could take raw web-page data and convert it into functional and executable test steps. We specifically reviewed mapping common user pathways (login to account, complete forms, navigate menus) that users typically follow when using websites normally.

6.2.5 Execution Component

We validated both static and dynamic ai test generators to ensure they could take raw web-page data and convert it into functional and executable test steps. We specifically reviewed mapping common user pathways (login to account, complete forms, navigate menus) that users typically follow when using websites normally.

6.2.6 Coverage and Export Component

We validated both static and dynamic ai test generators to ensure they could take raw web-page data and convert it into functional and executable test steps. We specifically reviewed mapping common user pathways (login to account, complete forms, navigate menus) that users typically follow when using websites normally.

6.3 Unit Testing

We ran unit testing in order to ensure that every single piece of our program works as intended for all of its functions through out the whole application. We used the same utilities we would use to test everything listed previously; logic that determines whether an answer is a pass or fail depending on how good it is at meeting the requirements; repository functionality (to add, modify, retrieve); to parse URL's and get schema info as well as to normalize the test steps.

Here are several examples of things that were evaluated during unit testing:

* Properly validating and formatting the web address;
* Logic for determining Pass/Fail rules for Test Results;
* Determining which fields in the schema require user input;
* Add data to Modify Retrieve Data from live repository;
* Create Node Map with edges for coverage map;
* Convert raw test data from repository to steps to display on screen;

Thus we were able to verify that each component worked correctly prior to combining them into the full application.

6.4 Integrated Testing

We did Integration testing to verify that the independently developed components worked together correctly follow are the points tested in the project:

- Angular frontend with Electron IPC communication
- Electron controller with extraction module
- Extraction output with static and dynamic test generation
- Test generation output with execution module
- Execution results with evaluator and repository

- Repository and coverage data with frontend visualization
- Result data with export functionality

The integrated workflow was tested by submitting a target URL and observing whether the system completed all major stages in sequence.

6.5System Testing

The fully-finished version of the application was tested through System Testing to determine whether or not the application actually works well. Additionally, we needed to assess the difficulty of using the application, to establish how stable the application would remain when being used, and to validate how the application will work like an actual working application. In doing so, we took a closer look at the following areas:

6.5.1 Functional Validation

A comprehensive review of the entire platform has been completed to validate that it satisfies all of the main goals established previously. Therefore, we have confirmed each major process (adding a URL, pulling the interface data, creating the test(s), asking for the user's login credentials, performing the browser actions(s), validating the results, producing the coverage map, and exporting the final report) were validated.

6.5.2 Usability Validation

The front-end of the application has been validated to ensure that users of various skill sets can utilize the application in an easy manner. More specifically, we concentrated on confirming that all components of the application are visibly apparent and accessible from both screens (dual-screen mode), that live progression is presented to the user through each stage of execution, that the dialogue between the user and the AI chatbot is presented as it happens, and that all aspects of the graph showing coverage are easily viewable.

6.5.3 Reliability Validation

Common usage scenarios and less than common edge cases were run against the software. Examples of these include running the software with URLs that will not work properly, simulating an error condition occurring while extracting data, failing

to request passwords where they are required, etc. This was done to illustrate that in case an error does occur (other than crashing/freezing) that the application will provide notification to the user indicating what happened.

6.5.4 Performance Validation

Common usage scenarios and less than common edge cases were run against the software. Examples of these include running the software with URLs that will not work properly, simulating an error condition occurring while extracting data, failing to request passwords where they are required, etc. This was done to illustrate that in case an error does occur (other than crashing/freezing) that the application will provide notification to the user indicating what happened.

6.6 Test Cases

To validate that the software executes its desired functions, we created and ran a number of realistic "real world" test cases for all of the key operations that Testify AI performs.

Table 6: Website Analysis and Test Generation

Test Case Name	Website Analysis and Automatic Test Case Generation
Objective	To verify that the system can analyze a target website and generate test cases automatically.
Input	Valid website URL
Precondition	The application is running and the user is on the system testing screen.
Steps	1. Type or paste a working website URL into the input field. 2. Hit submit to send the address to the system for processing. 3. Give it a moment while the tool scans the page and pulls out the relevant elements. 4. Watch the chat panel as the system explains its thinking and lays out the planned test steps.
Expected Result	The application needs to scan the target webpage, pull out the key interactive components, create both standalone and workflow-

	driven test scripts, and show live status updates along with its reasoning steps directly on the screen.
Actual Result	The system successfully initiated extraction, identified interactive elements, grouped them into flows, and generated test case information for execution.
Status	Pass

Table 7: Runtime Credential Input and Test Execution

Test Case Name	Runtime Input Handling and Automated Execution
Objective	To verify that the system requests required input values and resumes execution correctly.
Input	Website URL containing authentication or input-dependent workflow
Precondition	Once an application is tested with random data (e.g. "generated"), the next step is to take the generated data from the users' real accounts and insert it into the testing application so that the test will operate as if the users were entering their own account information.
Steps	1. Run a test of a workflow which is dependent upon some user entry or input. 2. Wait until the system asks you to enter the appropriate values for the schema information required by each field in the interface's input fields. 3. Enter the required information in all of the input fields displayed in the interface. 4. Send this information back to the test engine. 5. Monitor the browser display to continue viewing the remainder of the test as it continues running automatically with your supplied data.
Expected Result	The system should request the required values, merge them into the test case, resume execution, perform the browser actions, and evaluate the result.

Actual Result	The system requested the values through the chat interface, accepted the user input, continued execution, and completed the test workflow successfully.
Status	Pass

Table 8: Coverage Graph Generation

Test Case Name	Coverage Visualization
Objective	To verify that the system records page transitions and displays them in coverage view.
Input	A website with multi-page or multi-route navigation
Precondition	The test suite should include at least one case that triggers page navigation.
Steps	1. Run the generated tests on a workflow that involves moving between pages. 2. Wait for the testing process to complete. 3. Open the coverage view to review the results.
Expected Result	The system should display a graph of visited pages and their transitions, linked to executed test cases.
Actual Result	The coverage graph correctly displayed page nodes and transitions generated during test execution.
Status	Pass

Table 9: Export of Test Results

Test Case Name	Test Report Export
Objective	To check that the system can save and export the final test results for record-keeping and documentation purposes.
Input	Completed testing session
Precondition	The test cases have already been created and run through the system.
Steps	1. Finish up the current test session. 2. Click the export button to generate the results file.

	3. Open the downloaded file to review the output.
Expected Result	The system should produce an Excel spreadsheet that summarizes all the test case outcomes, including pass/fail status and execution details.
Actual Result	The system exported the results into Excel format with relevant testing fields.
Status	Pass

6.7 Results & Evaluation

The test results show that Testify AI successfully delivered the core features outlined for the project. The system managed to:accept and process target URLs,

- extract interactive web page elements,
- generate static and dynamic test cases,
- request runtime user input where necessary,
- execute test steps in the browser,
- evaluate pass/fail outcomes,
- visualize navigation coverage,
- and export test summaries.

From an evaluation perspective, the project demonstrates that combining **desktop application design**, **browser automation**, and **AI-assisted test generation** can create an effective testing workflow for modern web applications.

We found several significant strengths during the course of our evaluation of Testify AI:

1. **Automation of multiple testing stages**
   Testify AI will do many things more than just run simple browser automation scripts (the AI can extract the information off the webpage, create your test cases, evaluate the test cases you created and generate your final report) and accomplish them in an efficient manner.
2. **Interactive and practical workflow**
   Users will have access to split screen views so they may view what is happening within their browser at the same time as viewing what the AI thinks should happen.

3. **Support for real-world scenarios**

   Due to Testify AIs capability to request user input (including current password fields), it will allow for testing of real world login interfaces and forms.

4. **Meaningful coverage tracking**

   The visual coverage map will provide users with a clear understanding of which URLs or links were used in conjunction with one another during the evaluation process.

   However, there are two areas that Testify AI needs improvement:

1. The quality of tests generated by the AI is directly dependent upon the AI being able to properly identify and understand the structure of the website it is attempting to test.
2. If websites are continuously updating/refreshing content, then the chances of the AI correctly identifying and selecting the intended element decreases significantly.
3. Currently, the only area of deployment for Testify AI is on an end-user's local desktop environment, as opposed to company-wide deployments.

Although these limitations exist, our evaluation demonstrated that Testify AI was successful in achieving its goal of developing a complete, automated tool for testing Web pages.

6.8 Conclusion

In this chapter, we provided a detailed description of how we put Testify AI to the test. Our evaluation of Testify AI utilized a systematic approach. The systematic approach involved examining individual parts of Testify AI (individual parts), determining if the overall logic behind Testify AI functioned as designed, combining individual sub-modules together and lastly evaluating the full product. We ran numerous simulations of real-world scenarios to validate that Testify AIs key features work (extracting relevant data from a website, producing test cases, navigating through a browser window, showing a coverage map, etc.). Therefore, based on the results of our testing, Testify AI performs flawlessly as a sophisticated desktop-based Web testing application. Next Chapter is Chapter 2 - Project Wrap-up & Future Directions for Testify AI.

CHAPTER 7
CONCLUSION

Conclusion

A lot of the work for the creation of Testify AI has already been done. The primary reason to create Testify AI was to get around the long time consuming process of manual checking and the need for a great deal of coding for most forms of automation. So, Testify AI can find all the interface items on a page, develop tests based on those found items, execute the tests itself, determine the outcomes of those tests and display a coverage map -- all from inside one desktop application. We have also shown through our research that there is substantial benefit in using AI in software testing.

7.1Contributions

Testify AI satisfies each of the goals we outlined at the beginning of this project.

We created a working desktop testing solution. Thus, we fulfilled our purpose of creating a practical tool for smart web testing. Unlike many other solutions that either focus only on web browsers or rely only on scripted pure code, we built a full-featured application using Angular and Electron to enable users to fully take advantage of their desktop environments.

We also made sure that we included interactive UI/UX experience as well as automatic identification and definition of common user navigation routes. It identifies and maps site elements and defines logical navigation patterns enabling the software to recognize clickable objects and mimic how a user would normally click through live websites. Through this identification of site elements and defining logical pathways, the software sets up a base line for running automated tests.

We built a capability to self-generate test cases — which include both individual element-level tests as well as complex dynamic pathway tests. And we accomplished this without increasing reliance upon man written tests. Because it can generate successful scenarios as well as failure scenarios, the generated tests represent realistic world usage.

Finally, we included direct automated execution and evaluation within the browser. Thus, we addressed the goal of not only executing tests but also evaluating results. Using auto-grade functions to measure test successes/failures enables the software to accomplish far beyond just randomly clicking.

Additionally, we added features including password management for live logins, reporting tools for tracking test coverage, and export reporting tools. These enhancements make the software more usable in actual world applications like logging into secure login screens or determining what portions of a website were tested. This fulfills our goal of making the software practical, useful and applicable to real-world quality assurance efforts.

So, we accomplished our major objectives by building a sophisticated testing tool that will perform various tasks independently and seamlessly including; pulling data off sites, generating test cases, executing tests, evaluating test results, generating reports etc., etc.

7.2Reflections

The benefits of Testify AI are numerous in comparison to the benefits available from other testing options. For example, Testify AI recognizes that testing goes way beyond just clicking buttons. From start-to-finish — scanning pages, generating test cases, executing tests, evaluating results, and finally creating reports — it controls every step involved in the testing cycle. This broader range makes it more valuable than simple browser automation scripts.

Also, Testify AI has a unique desktop layout which allows users to see a live browser window simultaneously with the AIs' internal thought processes and a graphic visualization of test coverage. Over all this level of transparency is extremely beneficial especially in educational settings and when presenting this type of technology.

Lastly, Testify AI represents an ideal example of how smart automation can greatly reduce a large portion of the manual effort associated with testing thereby allowing testers to access websites in a more streamlined and efficient manner. Additionally,

Testify AI shows potential contributions to the growing field of AI-assisted testing — demonstrating how GUI scanning, browser robots and AI-generated test case generation can be combined into a working system.

7.3Future Work

The potential for this project to expand in various ways is vast.

One area where the project can grow significantly is to improve how often the data extractor extracts data from websites that are updated regularly. Some potential features for the next version of the data extractor might include self-healing, an intelligent locator that dynamically changes as needed, and additional features for extracting information from web pages that use extensive amounts of JavaScript.

Another way that the project can expand is by enabling the user to run tests in multiple browsers at once. If this were done, the project would go from being a very focused piece of software with limited uses to being a full-featured product that supports larger enterprises whose needs include ensuring cross-browser compatibility for all of their applications.

The third area for growth is integrating a database so that users can save their previous testing results, coverage reports, historical logs, and custom preferences. As users continue to run tests, they would have a record of what was previously tested. A permanent record of testing would make the tool much more useful and effective for long-term testing efforts.

A fourth area for growth involves integrating continuous integration / continuous deployment (CI/CD) pipelines into the application. This feature would allow developers to configure the application to automatically run tests after every commit of new code. The ultimate goal would be creating a complete, automated testing environment.

Future research directions may include making the comparisons between the AI's output and established benchmarks more rigorous. Because the AI output can vary depending on the size and type of dataset that was used to train the AI, having a

mechanism for determining how closely the generated tests align with actual business requirements through comparing them to standard industry benchmarks would give researchers a clear way to measure both the accuracy of the generated tests and therefore their utility.

Finally, there are many other types of testing that the application could potentially do beyond generating unit tests including UI checks on the layout of a website; running tests on mobile devices; checking API calls; and performing security scans. Using these capabilities in conjunction with existing tools and services could create an end-to-end Quality Assurance solution.

References

1. 1S. Dogan, A. Betin-Can, and V. Garousi, “Web application testing: A systematic literature review,” Journal of Systems and Software, vol. 91, pp. 174-201, May 2014, doi: 10.1016/j.jss.2014.01.010.

2. Banerjee, B. Nguyen, V. Garousi, and A. M. Memon, “Graphical user interface (GUI) testing: Systematic mapping and repository,” Information and Software Technology, vol. 55, no. 10, pp. 1679-1694, Oct. 2013, doi: 10.1016/j.infsof.2013.03.004.

3. M. Memon, M. E. Pollack, and M. L. Soffa, “Hierarchical GUI test case generation using automated planning,” IEEE Transactions on Software Engineering, vol. 27, no. 2, pp. 144-155, Feb. 2001, doi: 10.1109/32.908959.

4. M. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: Reverse engineering of graphical user interfaces for testing,” in Proc. 10th Working Conf. Reverse Engineering, 2003. [Online]. Available: https://www.cs.umd.edu/~atif/pubs/MemonWCRE2003-abstract.html. [Accessed: Apr. 13, 2026].

5. SeleniumHQ, “Selenium documentation overview,” Selenium Documentation. [Online]. Available: https://www.selenium.dev/documentation/overview/. [Accessed: Apr. 13, 2026].

6. Cypress.io, “Testing types,” Cypress Documentation. [Online]. Available: https://docs.cypress.io/app/core-concepts/testing-types. [Accessed: Apr. 13, 2026].

7. Microsoft, “Playwright introduction,” Playwright Documentation. [Online]. Available: https://playwright.dev/docs/intro. [Accessed: Apr. 13, 2026].

8. Microsoft, “Codegen,” Playwright Documentation. [Online]. Available: https://playwright.dev/docs/codegen. [Accessed: Apr. 13, 2026].

9. Testim, “AI test automation tool,” Testim. [Online]. Available: https://www.testim.io/test-automation-tool/. [Accessed: Apr. 13, 2026].

10. E. Ferreira, “About mabl,” mabl Help, Nov. 11, 2025. [Online]. Available: https://help.mabl.com/hc/en-us/articles/18967631285396-About-mabl. [Accessed: Apr. 13, 2026].

11. Applitools, “Autonomous testing platform,” Applitools. [Online]. Available: https://applitools.com/platform/autonomous/. [Accessed: Apr. 13, 2026].

12. Cypress.io, “Cypress Studio AI,” Cypress Documentation, Mar. 2026. [Online]. Available: https://docs.cypress.io/app/guides/cypress-studio. [Accessed: Apr. 13, 2026].

13. V. Garousi, A. B. Keleş, Y. Balaman, Z. Özdemir Güler, and A. Arcuri, “Model-based testing in practice: An experience report from the web applications domain,” Journal of Systems and Software, vol. 180, art. no. 111032, Oct. 2021, doi: 10.1016/j.jss.2021.111032.

14. M. Brunetto, G. Denaro, L. Mariani, and M. Pezzè, “On introducing automatic test case generation in practice: A success story and lessons learned,” Journal of Systems and Software, vol. 176, art. no. 110933, Jun. 2021, doi: 10.1016/j.jss.2021.110933.

15. X. Chang, Z. Liang, Y. Zhang, L. Cui, Z. Long, G. Wu, Y. Gao, W. Chen, J. Wei, and T. Huang, “A reinforcement learning approach to generate test cases for web applications,” in Proc. Int. Workshop Automation of Software Test (AST), 2023. [Online]. Available: https://conf.researchr.org/details/ast-2023/ast-2023-papers/10/A-Reinforcement-Learning-Approach-to-Generate-Test-Cases-for-Web-Applications. [Accessed: Apr. 13, 2026].

16. D. Amalfitano, R. Coppola, D. Distante, and F. Ricca, “AI in GUI-based testing: A survey of techniques, tools, and perceived advantages and limitations,” Journal of Systems and Software, vol. 235, art. no. 112751, 2026, doi: 10.1016/j.jss.2025.112751.

17. M. Tasarsu, A. V. Tokmak, and C. Catal, “Test case generation using large language models: A systematic literature review,” Cluster Computing, vol. 29, art. no. 227, 2026, doi: 10.1007/s10586-026-06021-z.

18. Y. Li, P. Liu, H. Wang, J. Chu, and W. E. Wong, “Evaluating large language models for software testing,” Computer Standards & Interfaces, vol. 93, art. no. 103942, Apr. 2025, doi: 10.1016/j.csi.2024.103942.

19. N.-K. Le, Q. M. Bui, M. N. Nguyen, H. Nguyen, T. Vo, S. T. Luu, S. Nomura, and M. L. Nguyen, “Automated web application testing: End-to-end test case generation with large language models and screen transition graphs,” 2025. [Online]. Available: https://arxiv.org/abs/2506.02529. [Accessed: Apr. 13, 2026].

APPENDIX A

Abbreviations

- **AI** — Artificial Intelligence
- **CI/CD** — Continuous Integration / Continuous Deployment
- **DOM** — Document Object Model
- **E2E** — End-to-End
- **FYP** — Final Year Project
- **GUI** — Graphical User Interface
- **IPC** — Inter-Process Communication
- **LLM** — Large Language Model
- **NFR** — Non-Functional Requirement
- **QA** — Quality Assurance
- **SDG** — Sustainable Development Goals
- **UML** — Unified Modeling Language
- **URL** — Uniform Resource Locator

APPENDIX B

Minimum System Requirements (Suggested)

- **Operating System:** Windows 10/11, macOS, or Linux (64-bit)
- **CPU:** Dual-core 2.0 GHz (recommended: 4 cores)
- **RAM:** 8 GB (recommended: 16 GB)
- **Disk:** 2 GB free space (plus space for exported reports)
- **Internet:** Required for testing online websites and (if used) external AI services
- **Browsers:** Chromium-based browser binaries installed by Playwright

APPENDIX C

Development Environment (Example)

- **IDE:** Visual Studio Code
- **Runtime:** Node.js (LTS)

- **Frontend:** Angular
- **Desktop Shell:** Electron
- **Automation:** Playwright
- **Export:** XLSX library
- **Version control:** Git

APPENDIX D

User Guide (Quick Start)

Open **Testify AI**.

Open up System Testing from the top left corner of your Dashboard. Type in the url you want to test and let it gather data. Read through what was gathered with the test plan created. If the tool requests live input (i.e. username/password) type that right into the prompt box.
Go ahead and run the test(s). View the test run results in real time in the browser window.
After all testing is complete review your overall success rate of test run v/s failure rate. Also view how each decision was made by viewing the logic applied during the test run.
Look at your coverage map so you know which paths/pages were tested. Export ALL data to an Excel Spreadsheet for future reference.

APPENDIX E

Functional Requirements Traceability

Table 10: Functional Requirement Traceability Table

FR ID	Description	Implemented In	Test Evidence
FR-01	URL submission	Frontend URL form + Electron controller	Chapter 6 – Test Case #1
FR-02	UI element extraction	Extraction module	Chapter 6 – Component Testing
FR-03	Static test generation	Static test generator	Generated test list screenshots/logs

FR-04	Flow-based test generation	Dynamic flow generator	Chapter 6 – Test Case #1
FR-05	Runtime input handling	Schema request + chat panel	Chapter 6 – Test Case #2
FR-06	Test execution	Playwright execution engine	Execution logs / screenshots
FR-07	Result evaluation	Evaluation module	Pass/Fail summaries
FR-08	Coverage visualization	Coverage manager + UI graph	Chapter 6 – Test Case #3
FR-09	Export results	XLSX export module	Chapter 6 – Test Case #4

APPENDIX F

Export Report Fields (Example)

- Test ID
- Priority (High/Medium/Low)
- Purpose
- Intent
- Steps summary (action + target)
- Status (Pass/Fail)
- Failure reason (if any)
- Timestamp

APPENDIX G

Saad Ilyas Pl Rep

ORIGINALITY REPORT

6%	5%	2%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

#	Source	%
1	Submitted to Higher Education Commission Pakistan Student Paper	2%
2	bahria.edu.pk Internet Source	1%
3	Submitted to Laureate Higher Education Group Student Paper	<1%
4	www.freepatentsonline.com Internet Source	<1%
5	www.coursehero.com Internet Source	<1%
6	Submitted to International University - VNUHCM Student Paper	<1%
7	Linda T. Miller, Philip A. Vernon. "Intelligence, reaction time, and working memory in 4- to 6-year-old children", Intelligence, 1996 Publication	<1%
8	d.lib.msu.edu Internet Source	<1%
9	repository.sgu.ac.id Internet Source	<1%

#	Source	Similarity
10	Submitted to Erasmus University of Rotterdam Student Paper	<1%
11	Submitted to University of Technology, Mauritius Student Paper	<1%
12	Submitted to National School of Business Management NSBM, Sri Lanka Student Paper	<1%
13	Submitted to Queen Mary and Westfield College Student Paper	<1%
14	Submitted to Islington College,Nepal Student Paper	<1%
15	Submitted to British University in Egypt Student Paper	<1%
16	Submitted to Infrastructure University Kuala Lumpur Student Paper	<1%
17	dl.ucsc.cmb.ac.lk Internet Source	<1%
18	Submitted to Technological University Dublin Student Paper	<1%
19	Submitted to CSU, San Jose State University Student Paper	<1%
20	kipdf.com Internet Source	<1%

www.pcgamefreetop.net

21	Internet Source	<1 %
22	Submitted to British Institute of Technology and E-commerce Student Paper	<1 %
23	Submitted to Informatics Education Limited Student Paper	<1 %
24	conf.researchr.org Internet Source	<1 %
25	spaculus.com Internet Source	<1 %
26	theses.ncl.ac.uk Internet Source	<1 %
27	www.une.edu.au Internet Source	<1 %
28	Domenico Amalfitano, Riccardo Coppola, Damiano Distante, Filippo Ricca. "Chapter 23 AI inGUI-Based Software Testing: Insights fromaSurvey withIndustrial Practitioners", Springer Science and Business Media LLC, 2024 Publication	<1 %
29	d197for5662m48.cloudfront.net Internet Source	<1 %
30	ijctjournal.org Internet Source	<1 %
31	scholar.ucu.ac.ug Internet Source	<1 %

32	www.mdpi.com Internet Source	<1 %
33	www.sejournal.net Internet Source	<1 %
34	Sheghembe, Bakari Said. "Exploring Application of Radio Frequency Identification in log Tracking and Monitoring Systems: A Case of Sao Hill Forest Plantation", University of Dodoma (Tanzania) Publication	<1 %
35	e-archivo.uc3m.es Internet Source	<1 %
36	export.arxiv.org Internet Source	<1 %
37	ferhat.ai Internet Source	<1 %
38	gupea.ub.gu.se Internet Source	<1 %
39	ir.unisa.ac.za Internet Source	<1 %
40	kc.umn.ac.id Internet Source	<1 %
41	lutpub.lut.fi Internet Source	<1 %
42	repository.usd.ac.id Internet Source	<1 %
43	tel.archives-ouvertes.fr Internet Source	

<1 %

44	www.readbag.com Internet Source	<1 %
45	www.thinkmind.org Internet Source	<1 %
46	Iva Kertusha, Gebremariam Assres, Onur Duman, Andrea Arcuri. "A Survey on Web Testing: On the Rise of AI and Applications in Industry", Science of Computer Programming, 2026 Publication	<1 %
47	Jayesh Umre, Ashish Singh Parihar, Atul Gupta. "CSLLM: Code-Specific Large Language Models-A Survey", Expert Systems with Applications, 2026 Publication	<1 %
48	Yuan, X.. "Iterative execution-feedback model-directed GUI testing", Information and Software Technology, 201005 Publication	<1 %

Exclude quotes	On	Exclude matches	< 4 words
Exclude bibliography	On		

APPENDIX H

Submission ID trn:oid:::1:3539331018

*% 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.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.