PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research

UNIVERSITY OF ECHAHID HAMMA LAKHDAR - EL OUED

FACULTY OF EXACT SCIENCES
Computer Science Department

Thesis
submitted in partial fulfilment of the requirements for the degree of

ACADEMIC LICENCE

Field: **Mathematics and Computer Science**
Option: **Computer Science**
Specialty: **Information Systems**
Submitted by : **Mohieddine Matallah**

Title

Developing a Code Review Strategy Using LLMs and Acceptance Tests in Python

Defended on June 16, 2025

Examination Commitee :

M.	.....	Professor	Chair
M.	.....	MCA	Examiner
Dr. Mohammed Mounir BOUHAMED		MCB	Supervisor
Dr. Ahmed GHENABZIA		MCB	Supervisor

Academic year: 2024-2025

Acknowledgment

First and foremost, I express my deepest gratitude to Allah for granting me the health, strength, and perseverance to complete this modest work.

To my beloved father, who tirelessly strives day after day that I may drink from the fountain of knowledge and ascend the ranks of scholarship. To my precious mother, who spent sleepless nights praying for me, offering endless care and affection. To all who stood by my side, supporting me near or far, even with but a kind word.

I would like to express my sincere gratitude to my supervisors, Dr. BOUHAMED Mohammed Mounir and Dr. GHNEBZIA Ahmed, for their valuable support and guidance throughout this project. Dr. BOUHAMED provided insightful advice and consistent direction, which were essential to the successful completion of this work. I am also deeply thankful to Dr. GHNEBZIA for his technical assistance and generous mentorship, which greatly helped in addressing the challenges faced during the implementation phase.

To my honorable professors and every mentor who taught me even a single letter of knowledge - your wisdom lights the path of learning.

Abstract

In agile software development, the manual process of linking user stories to test coverage is slow and error-prone. This process creates significant bottlenecks, delays release cycles, and increases the risk of software defects. To address this problem, we developed a tool that automatically generates Pytest tests from user stories written in natural language. The tool leverages natural language processing (NLP) to translate non-technical requirements into executable code. First, the tool analyzes a user story to generate proposed acceptance criteria. A user then reviews and confirms these criteria. Upon confirmation, the tool generates executable Python test functions to validate the expected behavior. This automated framework reduces manual effort, enhances test consistency, and enables faster, more reliable software development cycles. Our approach is particularly beneficial for agile teams that have limited dedicated testing experience, as it empowers them to maintain high standards of quality assurance.

Résumé

Dans le développement logiciel agile, relier les besoins des utilisateurs à une couverture de test adéquate est souvent long et sujet à des erreurs. Ce projet propose un outil intelligent qui génère automatiquement des tests Pytest à partir d'histoires utilisateur rédigées en langage naturel. Le système commence par générer des critères d'acceptation clairs à partir de l'histoire, que l'utilisateur peut modifier ou valider. Une fois ces critères confirmés, l'outil produit des fonctions de test en Python correspondant au comportement attendu. Grâce au traitement du langage naturel (NLP), cet outil facilite la transition entre les exigences fonctionnelles et les tests techniques. Il réduit l'effort manuel, améliore la qualité des tests et accélère le processus de développement, même pour les équipes ayant peu d'expérience en test logiciel.

ملخص

في تطوير البرمجيات باستخدام المنهجيات الرشيقة، تكون عملية ربط متطلبات المستخدم بتغطية اختبار فعّالة غالبًا بطيئة ومعرّضة للأخطاء. يهدف هذا المشروع إلى تطوير أداة ذكية تقوم تلقائيًا بإنشاء اختبارات انطلاقًا من قصص المستخدم المكتوبة بلغة طبيعية. يبدأ النظام بإنشاء معايير قبول واضحة بناءً على القصة، ويمكن للمستخدم مراجعتها وتعديلها حسب الحاجة. بعد التأكيد، يتم استخدام هذه المعايير لتوليد دوال اختبار بلغة بايثون تعكس السلوك المطلوب. باستخدام تقنيات معالجة اللغة الطبيعية، تساعد الأداة على تحويل المتطلبات المكتوبة إلى اختبارات تقنية بشكل دقيق. هذا يسهم في تقليل الجهد اليدوي، وتحسين جودة الاختبارات، وتسريع وتيرة تطوير البرمجيات حتى مع فرق ذات خبرة محدودة في مجال الاختبار.

Table of Contents

List of Figures

List of Tables

General Introduction

0.1 Background and Motivation

The rapid evolution and increasing complexity of software systems present significant challenges to traditional quality assurance paradigms. Driven by agile and DevOps methodologies, development cycles demand the continuous and rapid delivery of stable, secure software. Within this context, conventional testing methods, particularly manual testing, have become critical bottlenecks [1, 2]. These legacy approaches are frequently time-consuming, prone to human error, and scale poorly. Consequently, they are fundamentally incompatible with the pace of modern software engineering.

This inefficiency leads to severe consequences, including the late detection of bugs, delayed product releases, and a subsequent increase in remediation costs [3, 4]. The failure to ensure consistent performance and quality not only hinders development velocity but also undermines user trust and system reliability. Consequently, sophisticated automated testing is now essential for building and delivering modern software.

0.2 Contribution of This Thesis

This thesis presents a novel, AI-driven platform designed to address these critical limitations by automating and enhancing the software testing lifecycle. The platform provides an integrated solution that directly targets the inefficiencies of traditional methods. Its core contributions are:

- **Instantaneous code analysis** to identify potential defects, security vulnerabilities, and deviations from best practices upon submission.
- **Intelligent remediation suggestions** that are targeted and context-aware to enhance code quality, maintainability, and performance.
- **Automated generation of comprehensive test cases** to ensure robust functional coverage.
- **Detailed, real-time analytics** on performance metrics and quality benchmarks, delivered through an accessible web-based interface.

0.3 Contribution Objectives

The central goal of this contribution is to create a system that makes software testing more intelligent, efficient, and integrated. To achieve this, the work pursues the following specific objectives:

1. To conduct a systematic analysis of traditional and modern software testing methodologies to identify their primary limitations and define the requirements for an AI-driven solution.
2. To design a comprehensive system architecture for the intelligent testing platform using the Unified Modeling Language (UML). This design will specify the system's core components, data workflows, and integration points.
3. To develop a functional prototype of the proposed platform. This development will include implementing the AI engine for code analysis and test generation and constructing the web-based user interface.
4. To validate the platform's effectiveness against traditional methods. This validation will involve evaluating its performance in identifying defects and generating test cases.

0.4 Memory Outline

The remainder of this memory is organized as follows:

Chapter 2: Literature Review provides a detailed review of existing software testing techniques, discusses the evolution of test automation, and surveys current applications of artificial intelligence in quality assurance. This chapter establishes the theoretical foundation for the research.

Chapter 3: System Design and Architecture presents the detailed architectural framework for the AI-driven platform. It describes the system's components, the design of the AI models, and the overall workflow, supported by UML diagrams.

Chapter 4: Implementation details the development process of the platform. This chapter covers the technologies used, the implementation of the code analysis and test generation modules, and the construction of the user interface.

Chapter 5: Evaluation and Results presents the experimental validation of the platform. It outlines the methodology used to assess the system's performance and discusses the results, comparing its effectiveness against baseline testing methods.

Chapter 6: Conclusion and Future Work summarizes the key findings and contributions of the thesis, reflects on the achievement of the research objectives, and proposes potential directions for future research.

Chapter 1

Basic Concepts and Problem Analysis

1.1 Introduction

This chapter dives into the fundamental Basic Concepts and the crucial phase of Problem Analysis, setting the stage for everything that follows. Here, we'll establish a solid understanding of the core ideas and accurately break down the problem we aim to solve, In this chapter we cover the following components:

- Basic Concepts
- Review of Similar Code Review
- The Problem State

1.2 Basic Concepts

1.2.1 Code Reviewing

Code review is a standard practice in software engineering, defined as the critical examination of source code before its integration into a main branch [7]. This examination identifies bugs, logical errors, and potential edge cases. It also provides a second opinion on the proposed solution. The practice helps development teams identify security flaws, adhere to established quality standards, and share knowledge across the organization. Reviewers are typically domain experts from relevant teams; if a change spans multiple domains, an expert from each is expected to participate in the review.

Importance of Code Reviews

A robust code review process is widely recognized for fostering continuous improvement and preventing unstable code from reaching customers. This process should be integrated directly into a team's development workflow, requiring a teammate to examine all code prior to deployment.

Furthermore, code review functions as a mechanism for disseminating knowledge across an organization. For instance, a 2022 industry survey reported that 76% of developers consider code reviews "very valuable" [1].

Benefits of Code Reviews

The literature and industry best practices identify several key benefits of systematic code review:

- **Knowledge Sharing:** Reviewing code exposes developers to new techniques and solutions. Junior developers, in particular, learn from seniors in a manner similar to pair programming. This dissemination of knowledge prevents individuals from becoming single points of failure, ensuring operational continuity when a team member is on leave.
- **Early Bug Detection:** Code reviews facilitate the discovery and correction of bugs before a software release. Early detection allows developers to fix issues while the implementation context is still fresh; conversely, delaying fixes increases the cognitive load required to recall the code's logic. This early detection is also highly cost-effective.
- **Compliance and Standardization:** Given that developer backgrounds influence coding styles, code reviews are essential for enforcing shared standards. This enforcement is particularly critical for open-source projects with numerous contributors, where maintainers review all changes before merging them.
- **Improved Code Quality:** By enforcing coding rules and styles, code reviews promote consistency and simplify the long-term management of code. This collaborative process improves overall code quality as developers learn from one another, leading to more robust and maintainable systems over time.

1.2.2 Automation Testing

Automated testing is a software verification approach that uses specialized tools to execute test cases automatically. It is particularly well-suited for large-scale projects or for regression testing where tests are highly repetitive. While it complements, rather than replaces, manual testing, automation allows human testers to focus on higher-value exploratory tasks. Although automated testing introduces the overhead of test script maintenance, its adoption generally increases application quality, test coverage, and scalability [2].

1.2.3 Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development methodology in which developers write automated unit tests before writing the corresponding production code [5]. These tests effectively define the requirements for a new feature. The developer then writes the minimum amount of code

necessary to pass the tests. TDD is structured as an iterative cycle of test creation, implementation, and refactoring, commonly known as the Red-Green-Refactor cycle.

- **Red Phase:** A developer writes a test for a desired feature or behavior, which is expected to fail initially because the feature does not yet exist.
- **Green Phase:** The developer writes the minimal functional code required to make the failing test pass.
- **Refactor Phase:** The code is then refactored to improve its design, readability, and performance, ensuring that the test continues to pass.

1.2.4 User Stories in Software Engineering

In Agile software development, user stories are a primary tool for requirement specification [6].

- **Definition:** A user story is a brief, non-technical description of a software feature articulated from the perspective of an end user. It captures what the user wants to achieve and why, helping development teams to understand user needs and prioritize work.
- **Typical Format:** The conventional format for a user story is: As a [type of user], I want [some goal], so that [reason/benefit].
- **Example:** As a developer, I want to receive instant feedback on my code after pushing changes, so that I can fix issues early and improve code quality.

1.2.5 Acceptance Criteria

- **Definition:** Acceptance criteria are the predefined conditions that a software feature must meet to be considered complete. They define the ”definition of done” for a user story and ensure the feature behaves as expected by stakeholders.
- **Example (Gherkin Format):**

  Given I am on the login page When I enter valid credentials And I click the login button Then I should be redirected to the dashboard

1.2.6 Generative AI in Software Development

Generative AI

Generative Artificial Intelligence (AI) is a class of AI models capable of creating new, synthetic content based on patterns learned from existing data [4]. Unlike analytical AI, which focuses on classification or prediction, generative AI produces novel outputs—such as text, images, or code—in response to a given input or prompt.

Key Capabilities of Generative AI

The capabilities of modern generative models include:

- **Text Generation:** Writing articles, emails, summaries, and other forms of creative or technical text.
- **Image Creation:** Producing original art, illustrations, logos, and photorealistic designs.
- **Video Generation:** Creating synthetic video clips or animations from textual descriptions.
- **Audio/Speech Synthesis:** Generating human-like voice, music, or sound effects.
- **Code Generation:** Assisting in software development by generating, completing, or debugging source code.

1.2.7 Prompt Engineering

Prompt engineering is the discipline of designing and optimizing inputs (prompts) to elicit desired, accurate, and relevant outputs from a generative AI model. This process involves carefully structuring questions, instructions, and contextual information provided to the model.

1.3 Review of Similar Code Review Tools

To contextualize the problem space, this section reviews three prominent tools that represent the current state of automated and AI-assisted code review.

1.3.1 GitHub Copilot

GitHub Copilot, powered by OpenAI's Codex model, is an AI-powered tool that suggests code and entire functions in real-time within the developer's editor. It is designed to accelerate the coding process and help enforce best practices by reducing repetitive tasks. While primarily a code completion tool, its capabilities can indirectly aid the code review process by influencing the initial quality of the code being submitted.

Figure 1.1: GitHub Copilot Logo.

1.3.2 SonarQube

SonarQube is an open-source platform for continuous inspection of code quality. It performs static code analysis to detect bugs, code smells, and security vulnerabilities, providing dashboards and reports with actionable insights for improvement. SonarQube often integrates into CI/CD pipelines to enhance manual code reviews with automated quality gates.

Figure 1.2: SonarQube Logo.

1.3.3 CodeClimate

CodeClimate is an automated code review platform focused on maintaining long-term code quality and maintainability. It analyzes code for metrics such as cyclomatic complexity, readability, and test coverage, providing detailed feedback and quality scores on each commit.

Figure 1.3: CodeClimate Logo.

1.4 The Problem Statement

1.4.1 Analysis of Problems in Conventional Code Review

A critical analysis of conventional code review methods, both manual and automated, reveals several persistent challenges that limit their effectiveness [3].

Inconsistent Feedback Quality

The quality of manual review feedback is highly dependent on the individual reviewer's experience, diligence, and available time. This variability leads to inconsistent outcomes, where some reviews are thorough while others miss critical issues or focus on trivial stylistic matters.

Process Inefficiency and Delays

Manual code reviews are inherently time-consuming, particularly for large pull requests or when key developers are unavailable. These delays create bottlenecks in the development lifecycle. Consequently, teams sometimes rush reviews simply to unblock deployment pipelines, which results in low-quality feedback.

Limited Contextual Awareness

Many automated review tools analyze code in isolation, lacking a broader understanding of the application's architecture, business logic, or domain-specific coding conventions. This lack of context can lead to false positives or a failure to detect subtle but significant logical flaws.

Predominantly Surface-Level Analysis

Most automated tools excel at identifying issues related to syntax, formatting, and simple metrics like code complexity or duplication. However, they often fail to identify deeper problems such as performance bottlenecks, complex security vulnerabilities, or incorrect implementations of business rules.

Feedback Overload and Alert Fatigue

Automated tools can generate a high volume of warnings, many of which may be low-priority or purely stylistic. This noise makes it difficult for developers to distinguish critical issues from minor ones, leading to alert fatigue and a diminished trust in the tooling.

Lack of Collaborative Features

Many review platforms treat code review as a static, asynchronous task rather than an interactive discussion. They offer limited support for real-time feedback, collaborative problem-solving, or structured knowledge sharing around the proposed changes.

Complexity in Setup and Integration

Configuring static analysis tools, managing rule sets and permissions, and integrating them effectively into CI/CD workflows can be a complex and time-consuming process, especially in large organizations with diverse technology stacks. This complexity can act as a barrier to consistent adoption.

1.4.2 The Need for an Intelligent, Context-Aware Solution

The identified shortcomings of current code review practices underscore a clear and pressing need for more intelligent, context-aware solutions. The inconsistency, inefficiency, and surface-level nature of existing methods create a significant gap between the goals of code review and its practical outcomes.

Addressing these challenges requires a new class of tool that moves beyond simple static analysis. Such a solution must be capable of understanding the business context of code, detecting complex logical errors, and providing prioritized, actionable feedback. Furthermore, it should enhance, not hinder, team collaboration by facilitating focused discussion.

Therefore, this research proposes and evaluates a novel AI-powered framework designed to bridge this gap. By leveraging generative AI guided by carefully engineered prompts, the proposed system aims to automate the generation of context-aware feedback, identify deeper quality issues, and streamline the review process more effectively than current methods allow. This work directly addresses the problems of inconsistent, surface-level analysis and provides a foundation for more collaborative and intelligent software quality assurance.

1.5 Conclusion

We've now covered the Basic Concepts and the important step of Problem Analysis. This chapter has given us a solid understanding of key practices like Code Reviews, Automated Testing, and Test-Driven Development (TDD). We also learned how User Stories and Acceptance Criteria help define what we need to build.

Chapter 2

Modeling and Design

2.1 Introduction

After mastering the foundational concepts and dissecting the analysis in the previous chapter, this chapter focuses on system design with UML, In this chapter we cover the following components:

- System Modeling with Unified Modeling Language
- Core Architectural Diagrams

2.2 System Modeling with Unified Modeling Language (UML)

This thesis uses the Unified Modeling Language (UML) to model the system design. UML is a standard graphical language for specifying, visualizing, and documenting software systems. The application of UML in this project serves four key objectives:

- To structure the development process methodically.
- To ensure clear communication between analysis and implementation efforts.
- To decouple the system's abstract analysis from its concrete implementation.
- To facilitate an object-oriented architecture for both static and dynamic system views.

2.3 Core Architectural Diagrams

2.3.1 Selected UML Diagrams

Specifically, this study employs three core UML diagrams to capture the system's architecture and behavior:

- **Use Case Diagram:** To define the system's functional requirements and primary user-system interactions.

- **Sequence Diagram:** To detail the time-ordered interactions between system components.
- **Activity Diagram:** To illustrate the flow of control through the system's primary workflow.

2.3.2 Use Case Diagram: Actors and Interactions

The system's architecture involves two primary actors: the *User* and the *AI System*. Table 2.1 outlines their distinct roles and responsibilities within the system.

Table 2.1: System Actors and Their Designated Roles

Actor	Roles
User	Enter User Story
	Edit Scenarios
	Confirm Scenarios
	Generate Acceptance Criteria
AI System	Generate Pytest Tests

The overall user-system workflow is visualized in the Use Case diagram (Figure 2.1). The process initiates when the user inputs a user story, which prompts the AI system to generate acceptance criteria. The user retains control to edit and confirm these criteria before the system proceeds to generate the final Pytest tests.

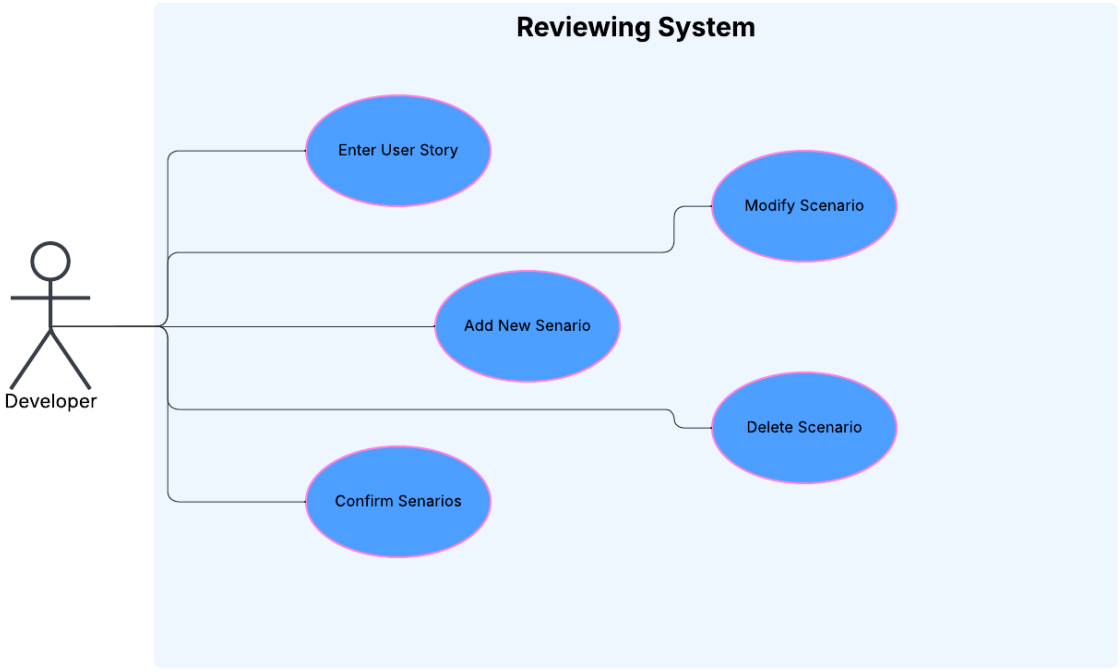

Figure 2.1: Use Case Diagram Defining User and System Actions.

2.3.3 Sequence Diagram: Component Interaction

The sequence diagram in Figure 2.2 illustrates the chronological interactions among the system's key components: the **Web Page** (front-end), the **Code** (back-end), and the **AI System**.

The sequence begins when a user submits a story on the **Web Page**. The back-end **Code** then validates the story. If the story is invalid, the **Web Page** prompts the user for correction. If it is valid, the **Code** invokes the **AI System** to generate acceptance criteria.

Once the user confirms these criteria, the **Code** directs the **AI System** to produce the Pytest tests. These tests are then rendered on the **Web Page**.

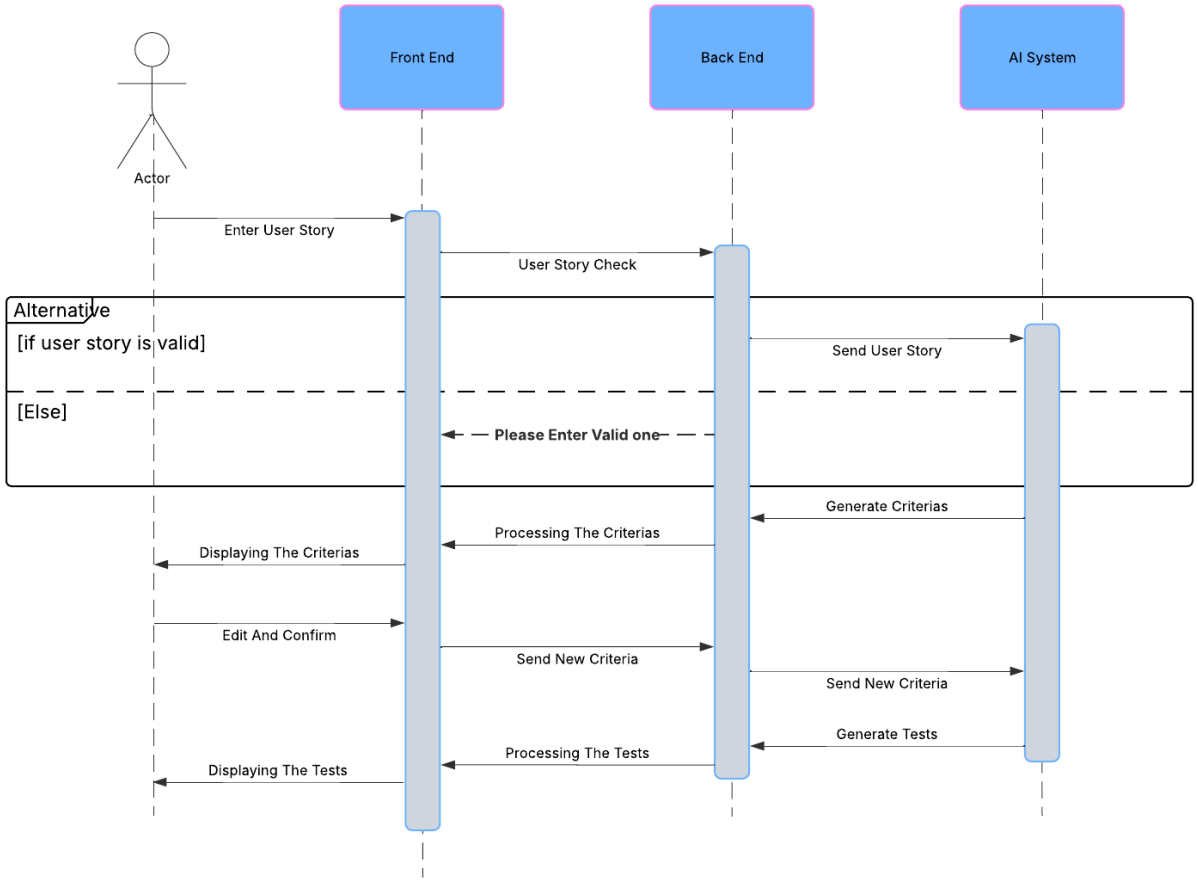

Figure 2.2: Sequence Diagram of System Component Interactions.

2.3.4 Activity Diagram: System Workflow

The activity diagram in Figure 2.3 details the complete workflow for automated test generation. This diagram uses swimlanes to delineate the responsibilities of the **User**, the **Web Page**, and the **AI System**.

The workflow commences when the **User** enters a user story. A decision node evaluates the story's validity. An invalid story prompts the **User** for modification. A valid story triggers the **AI**

System to generate acceptance criteria. Subsequently, the **AI System** generates the corresponding Pytest tests, which the **Web Page** displays to the **User**. The process concludes when the **User** copies the generated tests or chooses to restart the workflow.

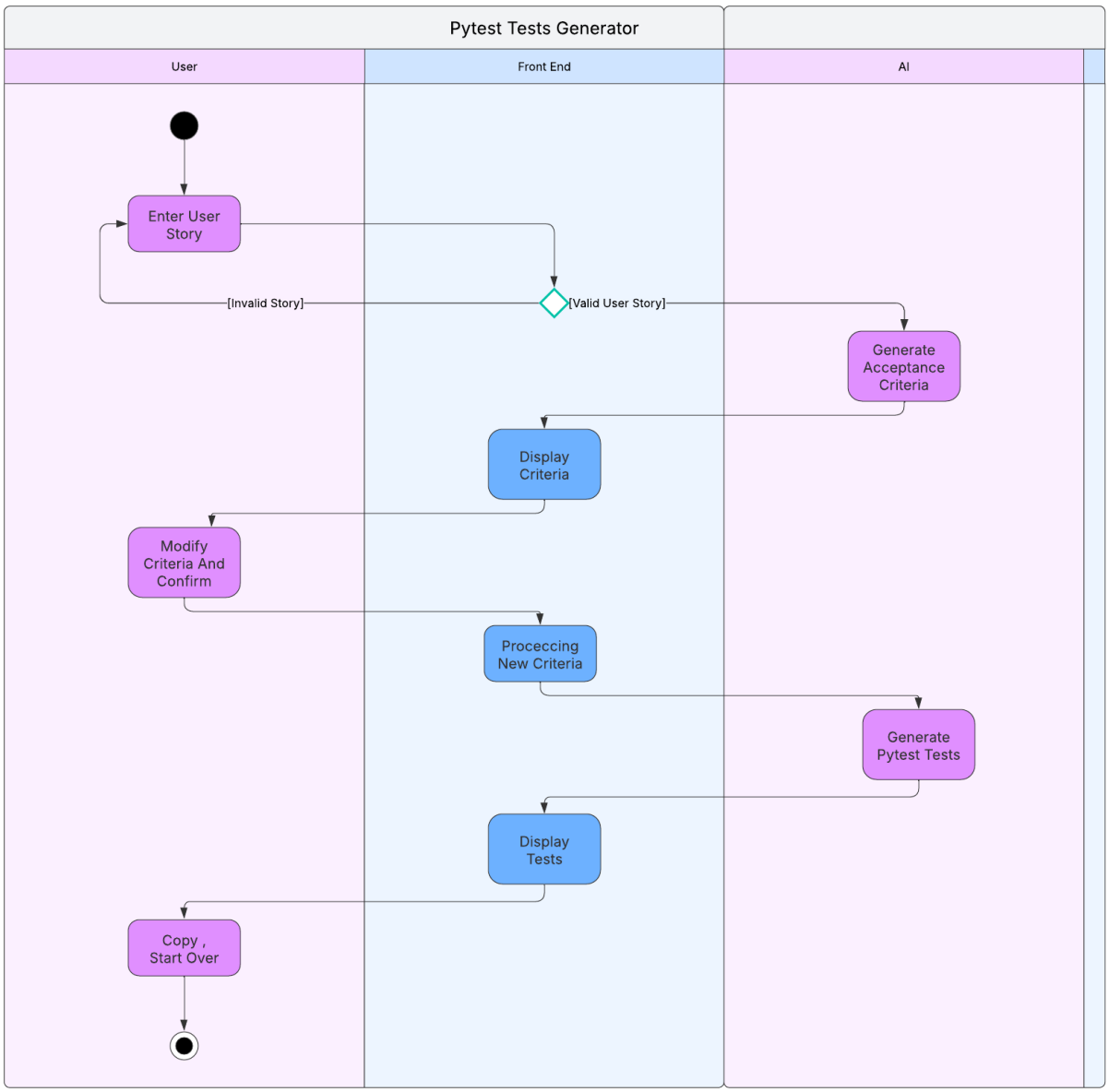

Figure 2.3: Activity Diagram of the Test Generation Workflow.

2.4 Conclusion

This chapter has successfully transformed our analytical understanding into a detailed blueprint for the system. Through the various UML diagrams, we have clearly defined the system's structure, behavior, and interactions.

Chapter 3

Framework

3.1 Introduction

This chapter details the system's implementation, outlining the development environment, programming languages, prompt engineering strategies, file structure, and application interfaces.

- Work Environment
- Programming Languages
- Prompt Engineering
- System Files
- Application Interfaces

3.2 Work Environment

3.2.1 Visual Studio Code

Visual Studio Code was selected as the primary source code editor for this project. It is an open-source editor from Microsoft, compatible with Windows, macOS, and Linux. Its native support for multiple programming languages prevented the need for separate, domain-specific editors.

The editor supports:

- Microsoft C
- Web development languages:
  - HyperText Markup Language (HTML)
  - Cascading Style Sheets (CSS)
  - JavaScript

- Various other programming languages

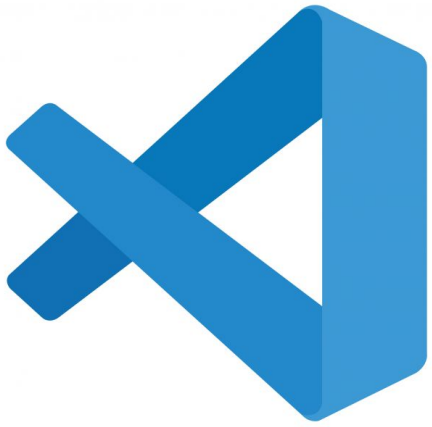

Figure 3.1: Vs Code Logo.

3.2.2 Gemini AI Studio

Overview

Gemini AI Studio was the development environment for the system's generative AI.

This Google cloud platform specializes in prompt engineering and model customization, providing tools for several key tasks:

- Interactive prompt design
- Multi-turn conversation management
- Output evaluation and refinement

Core Prompt Engineering Features

Gemini AI Studio provides several core prompt engineering features, including prompt optimization, collaboration, and advanced capabilities.

1. Prompt Optimization Tools

- **Template Gallery**: Pre-built prompts for common use cases
- **Autocomplete Suggestions**: Real-time prompt improvement recommendations
- **Safety Filters**: Built-in content moderation controls

2. Collaboration Features

- Version history for prompt iterations
- Team sharing with permission controls
- Commenting system for feedback

3. Advanced Capabilities

- **Multi-modal Prompts**: Combine text, images, and code
- **API Integration**: Deploy prompts via REST endpoints
- **Performance Analytics**: Track prompt effectiveness metrics

Supported Use Cases

Application	Prompt Example
Content Generation	”Generate 3 blog headlines about [topic]”
Data Analysis	”Extract key trends from this dataset...”
Code Assistance	”Explain this Python function line-by-line”

Table 3.1: Supported Use Cases Table

Best Practice: Gemini performs best when prompts include:

1. Clear intent specification
2. Relevant context
3. Desired output format

Figure 3.2: Gemini Logo.

3.3 Programming Languages

3.3.1 CSS (Cascading Style Sheets)

Cascading Style Sheets (CSS) was used to style the application's web pages. External stylesheets ensured consistent styling across the application.

Figure 3.3: CSS Logo.

3.3.2 HTML (HyperText Markup Language)

HyperText Markup Language (HTML) was used to structure the application's web pages. It defines content using semantic elements (e.g., headings, paragraphs) and forms the foundational layer of the user interface.

Figure 3.4: HTML Logo.

3.3.3 JavaScript

JavaScript was used to implement interactive web page functionality. As a client-side language, it handles user events, dynamically updates content, and communicates with the server. These capabilities were essential for creating a responsive user experience.

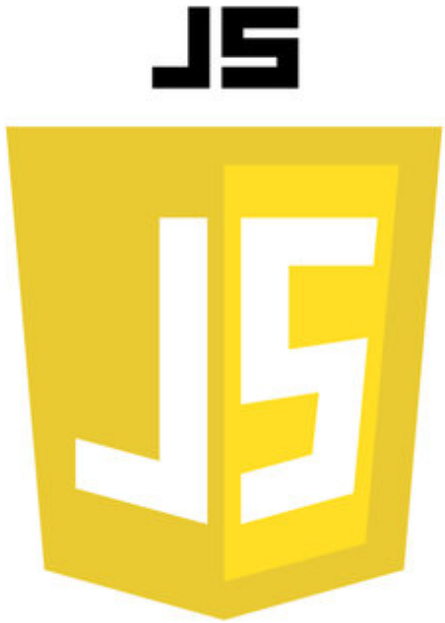

Figure 3.5: JavaScript Logo.

3.3.4 Python

Python was the primary language for server-side logic and AI integration. It was chosen for its readable syntax and extensive libraries, which facilitated backend processes, data management, and interfacing with the Gemini AI.

Figure 3.6: Python Logo.

3.4 Prompt Engineering

3.4.1 Acceptance Criteria Prompt Formulation

To guide the AI model in generating standardized acceptance criteria, a specific prompt was designed for this research. This prompt establishes strict requirements for the model's output.

Core Task

The primary task for the AI is to generate acceptance criteria for a given user story, following established Agile and Gherkin best practices.

Core Requirements for the AI Model

1. **Format Requirements:**
   - Must use the `Given-When-Then` structure for all criteria.
   - Each criterion must be atomic and independently testable.
2. **Content Requirements:**
   - Cover functional requirements (i.e., normal system behavior).
   - Include edge cases (e.g., invalid inputs, error conditions).
   - Address non-functional requirements where applicable (e.g., performance, security).
3. **Quality Standards:**

  - **Specific:** Define exact inputs, messages, and system behaviors.
  - **Observable:** Focus on user-visible outcomes.
  - **Unambiguous:** Avoid vague terms like ”properly” or ”correctly.”

Output Specifications

- Return only the acceptance criteria scenarios.
- Exclude any explanatory text or repetition of the user story.
- Format each scenario as a separate, distinct block.

Final Prompt Text

The following text constitutes the final engineered prompt provided to the model:

> ”You are an experienced business analyst in an agile environment. Your task is to write acceptance criteria for a user story. Follow these guidelines strictly:
>
> Format: Use ’Given-When-Then’ (GWT) for every criterion.
>
> Content: Include functional requirements, edge cases (e.g., invalid inputs, errors), and applicable non-functional requirements (e.g., performance, security).
>
> Specificity: Define exact inputs, messages, and system behaviors. Avoid vague terms.
>
> Atomicity: Ensure each criterion is independently testable.
>
> Perspective: Focus on user actions and observable outcomes.
>
> Provide only the acceptance criteria scenarios. Do not include explanatory text or repeat the user story.”

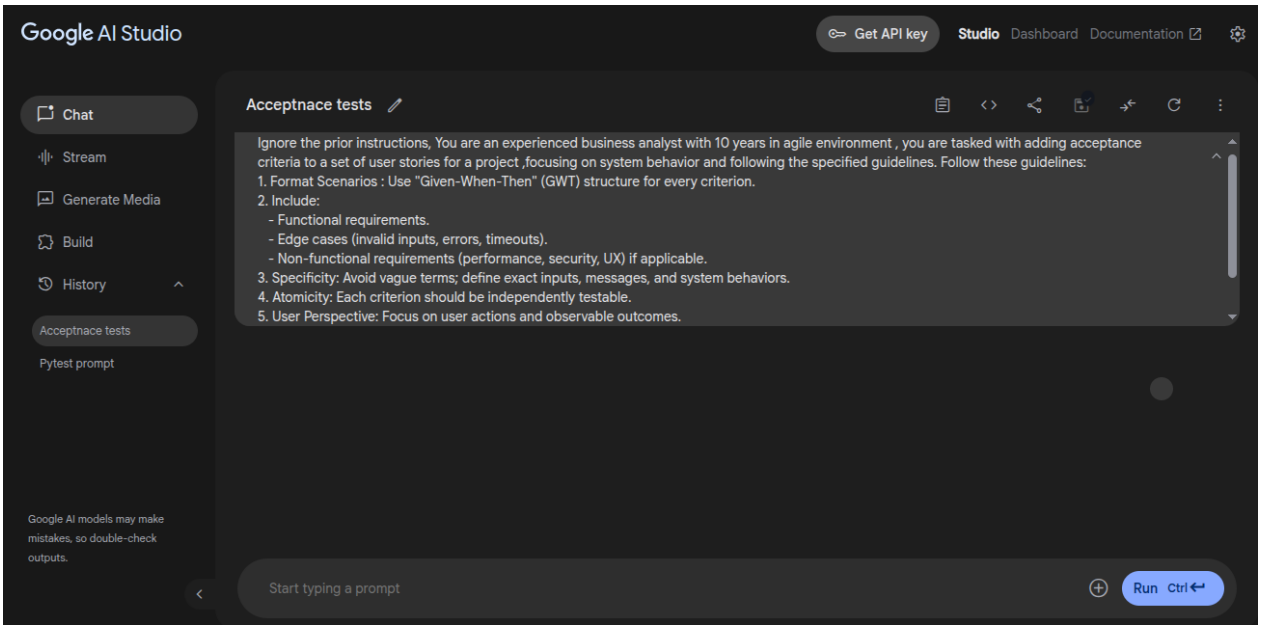

Figure 3.7: Acceptance Criteria Prompt.

3.4.2 Pytest Test Generation Prompt Formulation

A second prompt was engineered to instruct the AI model to generate Python test cases using the pytest framework.

Core Requirements for the AI Model

1. **Test Structure:**
   - Follow the `Arrange-Act-Assert` pattern, clearly delineated with comments.
   - Use pytest-style assertions (e.g., assert x == y) and avoid the unittest library.
   - Include clear, descriptive names for test functions.
2. **Code Quality:**
   - Employ pytest fixtures and mocks where appropriate to manage test setup and dependencies.
   - Focus on testing the relevant business logic.
   - Avoid unnecessary boilerplate code.
3. **Coverage:**
   - Generate tests for edge cases and error conditions in addition to the primary success path (”happy path”).
   - Validate all specified behaviors from the input scenario.

Output Specifications

- Return only complete, executable Python code.
- The output must be a single file ready to be run with pytest.
- Exclude any explanatory text outside of code comments.
- Format the code according to PEP-8 style guidelines.

Final Prompt Text

The following text constitutes the final engineered prompt provided to the model:

> ”Generate complete pytest test cases for the provided user story and scenario. Adhere to the following requirements:
>
> Structure: Use the Arrange-Act-Assert pattern with clear comments.
>
> Quality: Use descriptive test function names, pytest-style assertions (not unittest), and appropriate fixtures or mocks. Avoid boilerplate code.
>
> Coverage: Test the primary success path, error conditions, and relevant edge cases.
>
> Output: Provide only Python code in a single file, formatted according to PEP-8 and ready to be executed by pytest.”

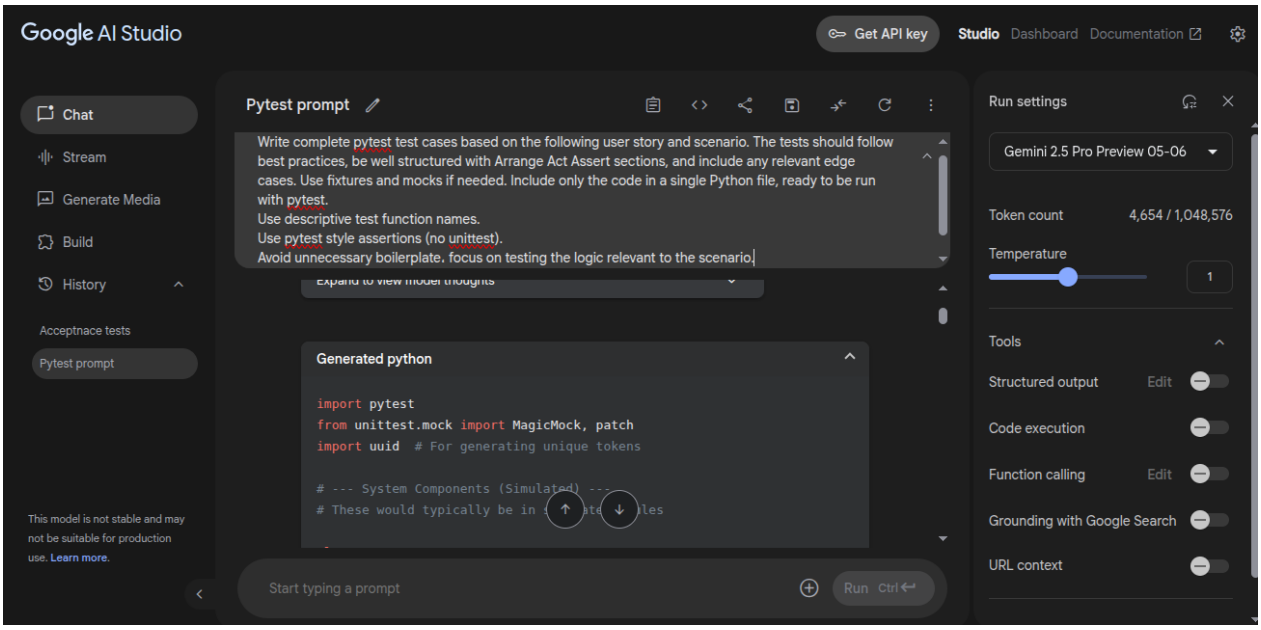

Figure 3.8: Acceptance Criteria Prompt.

3.5 System Files

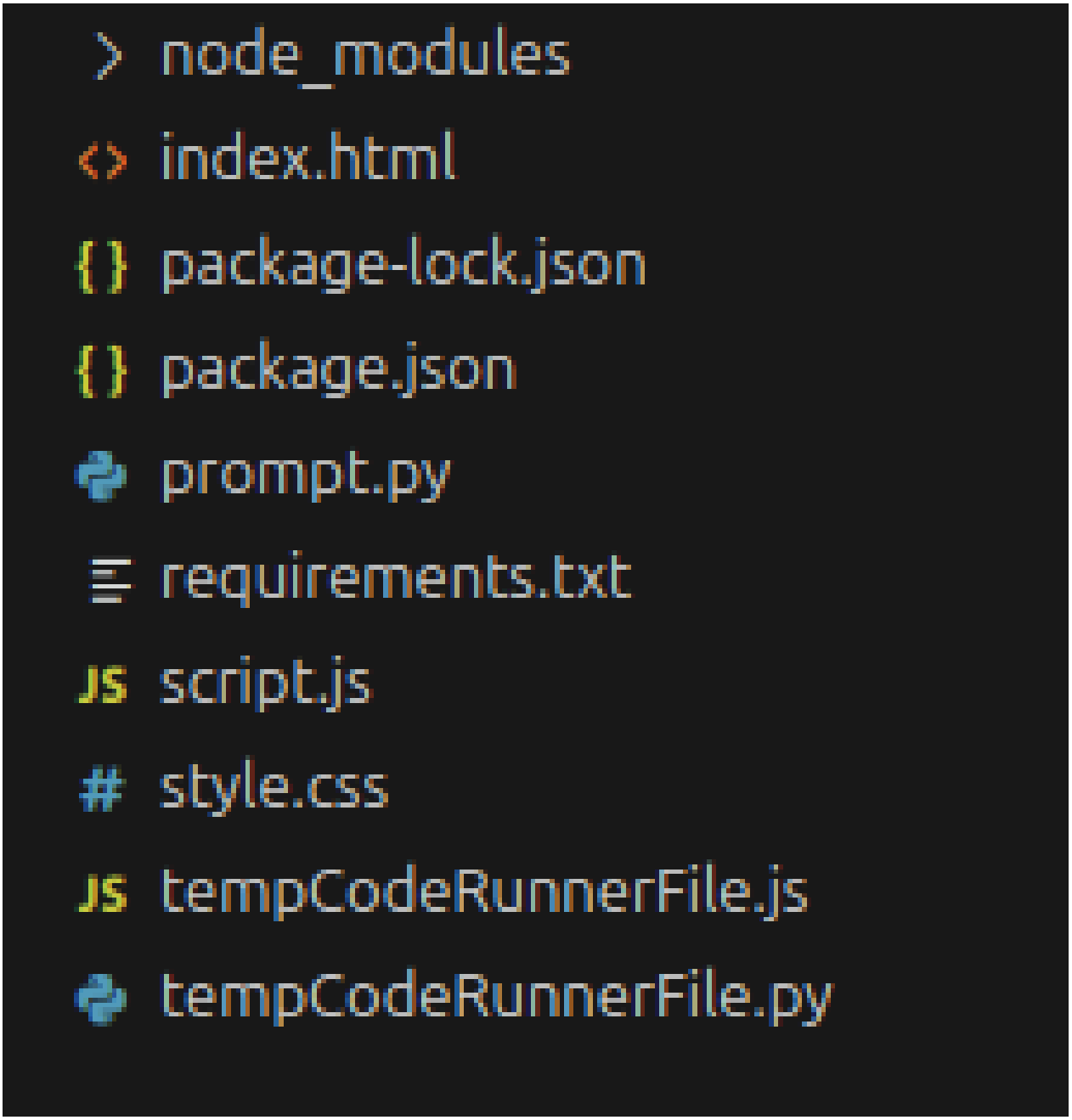

Figure 3.9: System File Structure.

3.6 Application Interfaces

3.6.1 User Story Interface

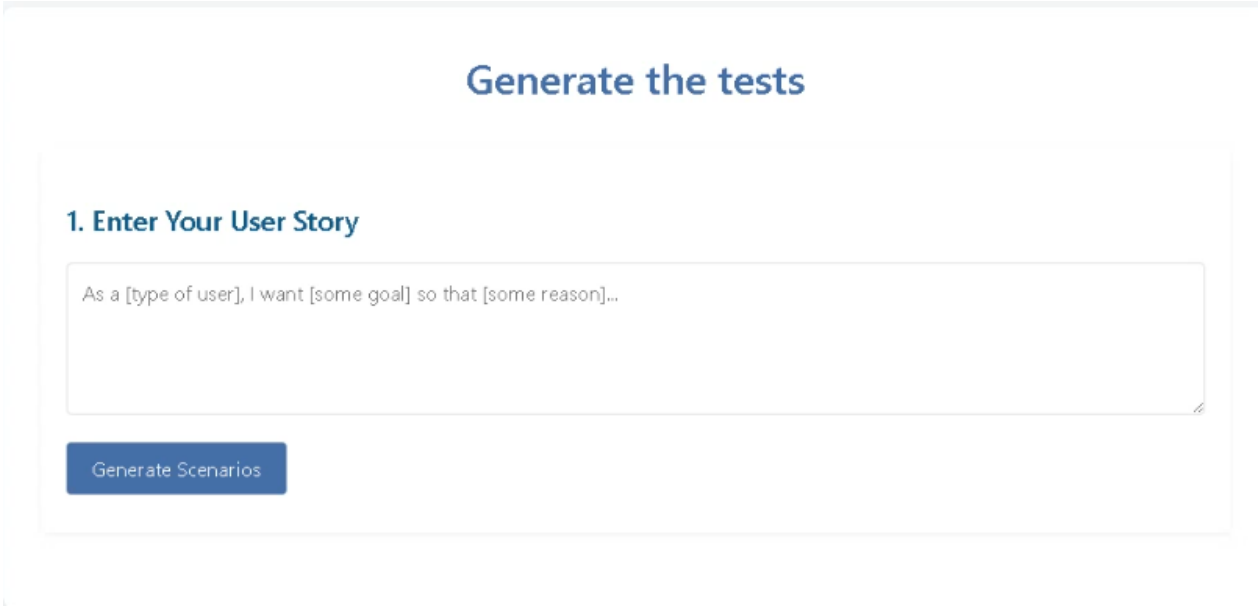

Figure 3.10: The User Story Input Interface.

3.6.2 Generated Acceptance Criteria

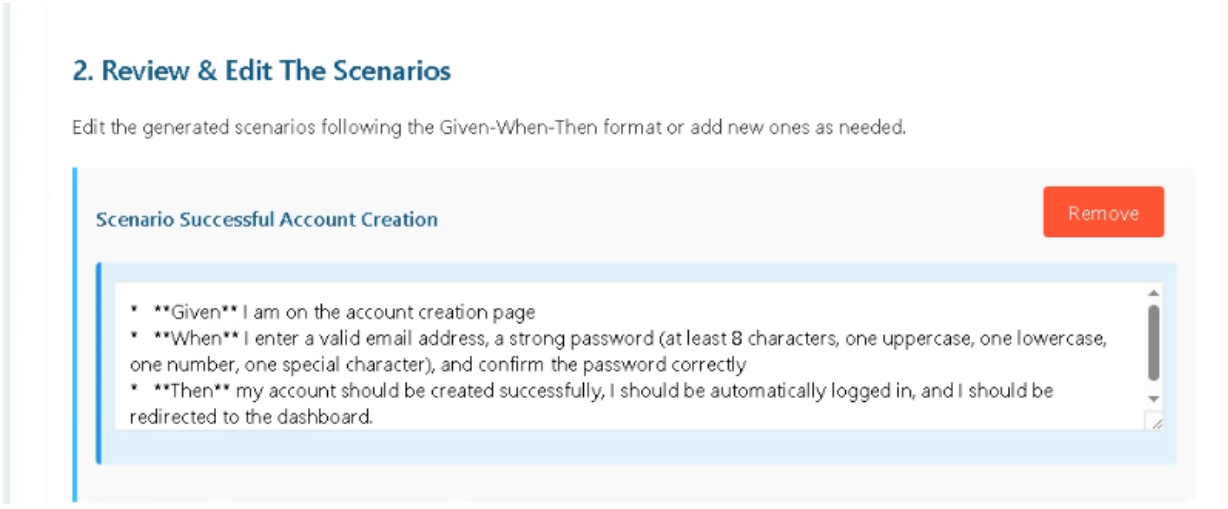

Figure 3.11: The Generated Acceptance Criteria Interface (Initial View).

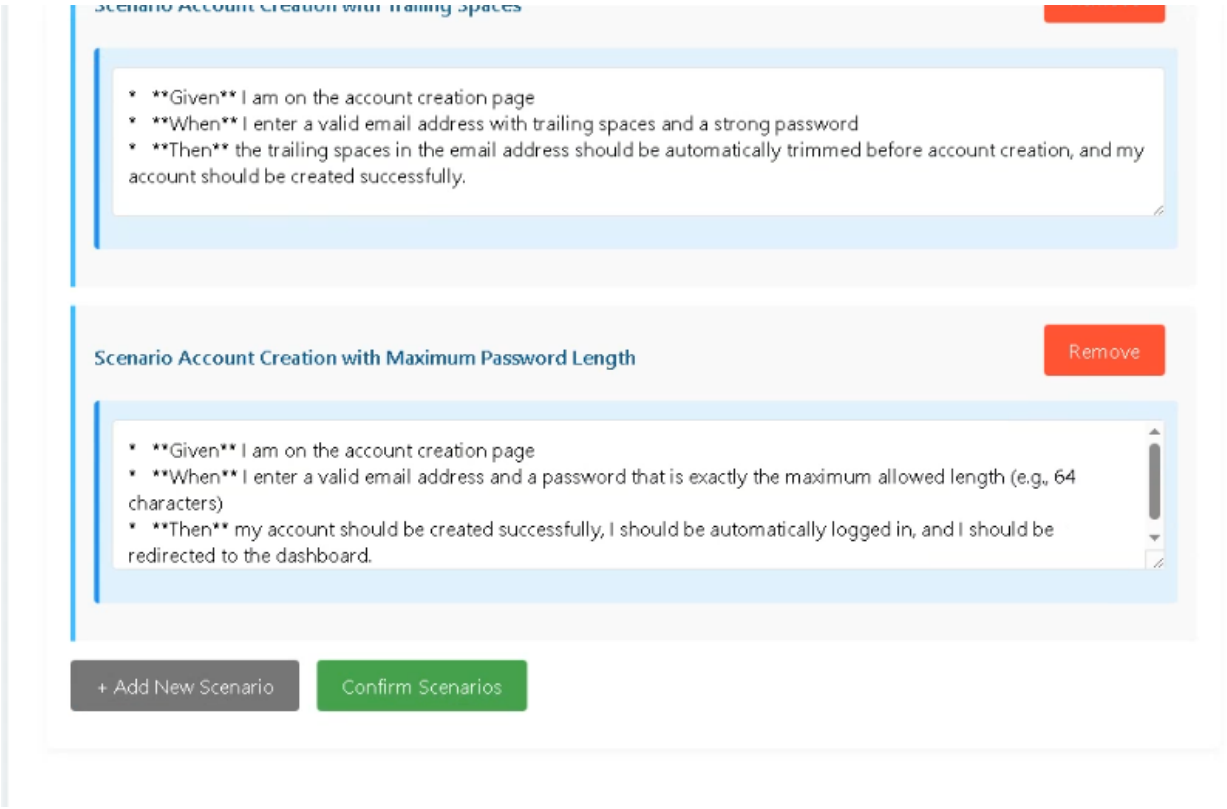

Figure 3.12: The Generated Acceptance Criteria Interface (Detailed View).

3.6.3 Generated Pytest Tests

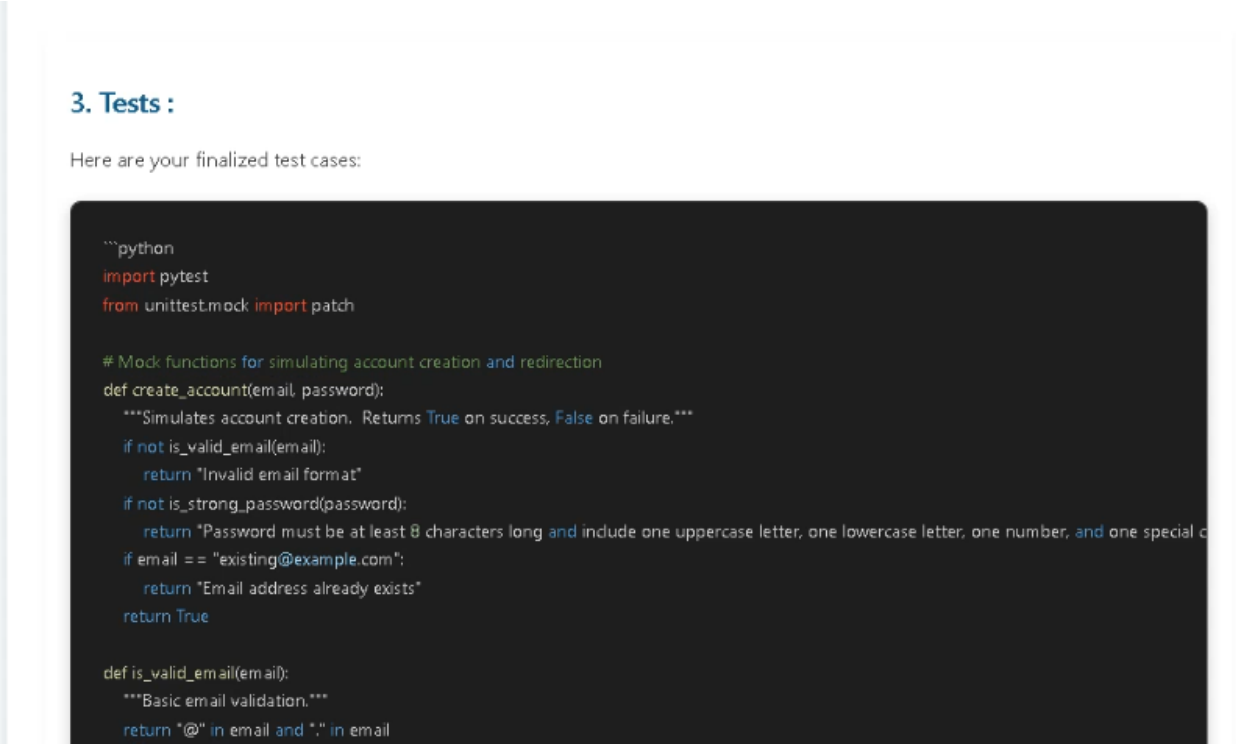

Figure 3.13: The Generated Pytest Tests Interface.

3.6.4 Pytest Test Execution

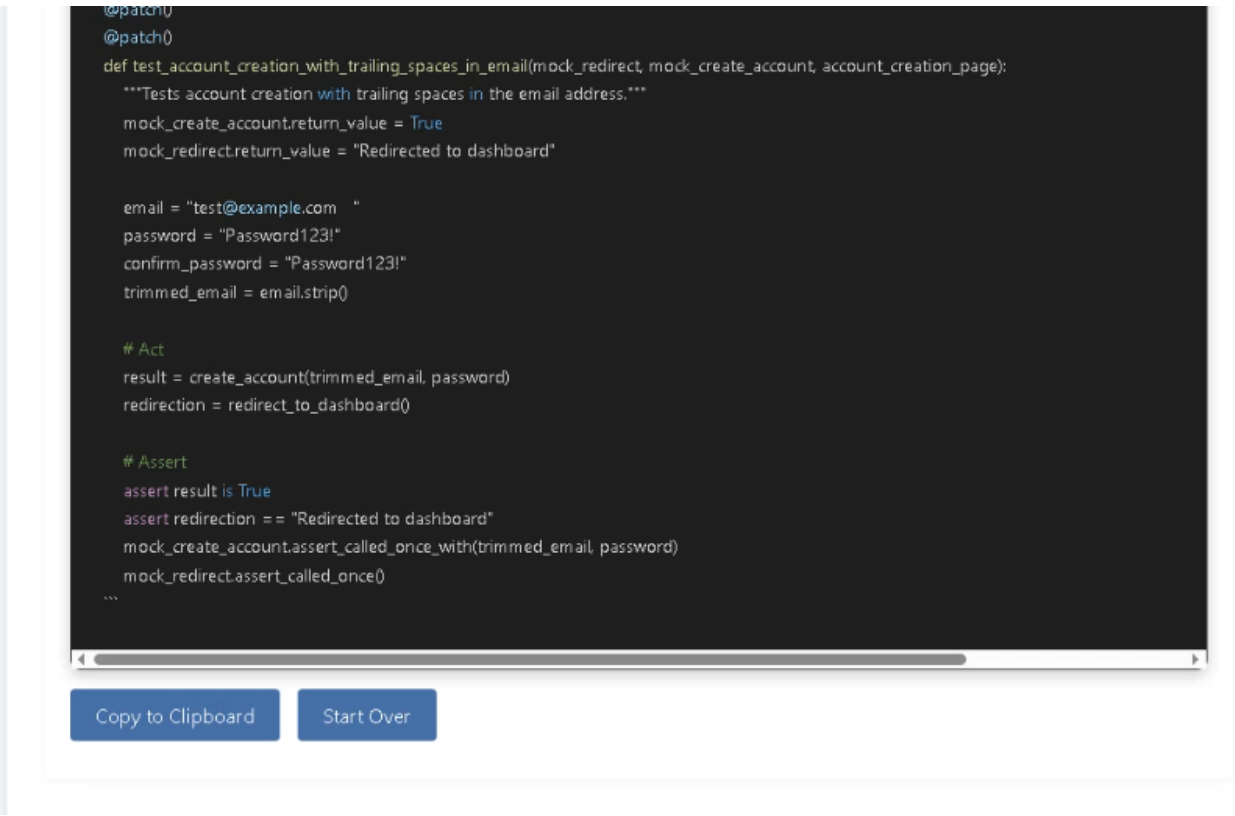

Figure 3.14: The Pytest Test Execution Interface.

3.7 Conclusion

This chapter detailed the technical framework for the intelligent test generation system. The development environment comprised Visual Studio Code and Gemini AI Studio. The application's front-end and back-end were built with CSS, HTML, JavaScript, and Python. The chapter also presented the system's file structure and key application interfaces.

General Conclusion

This document has guided us through a comprehensive journey, from foundational concepts to the detailed technical implementation of an innovative testing solution.

We began by establishing the Basic Concepts and Problem Analysis, exploring essential software engineering practices. This analytical groundwork then transitioned to the Design phase using UML, where the system's structure and behavior were precisely outlined. Finally, we addressed the Technical Environment, detailing the tools and languages critical for our work.

Crucially, this project showcases a novel approach to test automation. By leveraging Generative AI, we have demonstrated a streamlined workflow that transforms User Stories into structured Acceptance Criteria, which are then used to automatically generate Pytest tests. This approach significantly enhances efficiency and consistency in software quality assurance.

In conclusion, this work has laid a robust foundation, presenting a systematic progression from problem comprehension to a detailed, actionable plan. By integrating established methodologies with the power of Generative AI.

Bibliography

[1] What is code review? *gitlab*, 2023.

[2] A. Contan, C. Dehelean, and L. Miclea. Test automation pyramid from theory to practice. In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–5. IEEE, 2018.

[3] N. Fatima, S. Chuprat, and S. Nazir. Challenges and benefits of modern code review-systematic literature review protocol. In *2018 International Conference on Smart Computing and Electronic Enterprise (ICSCEE)*, pages 1–5. IEEE, 2018.

[4] S. Feuerriegel, J. Hartmann, C. Janiesch, and P. Zschech. Generative ai. *Business & Information Systems Engineering*, 66(1):111–126, 2024.

[5] S. Fraser, K. Beck, B. Caputo, T. Mackinnon, J. Newkirk, and C. Poole. Test driven development (tdd). In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 459–462. Springer, 2003.

[6] B. Meyer. Agile. *The good, the hype and the ugly. Switzerland: Springer International Publishing*, 2014.

[7] S. Nelson and J. Schumann. What makes a code review trustworthy? In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*, pages 10–pp. IEEE, 2004.