VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Implementing a Test Generation Service For Flutter Framework

By
Dao Minh Huy

A thesis submitted to the School of Computer Science and Engineering in partial fulfillment of the requirements for the degree of Bachelor of Computer Science

Ho Chi Minh City, Vietnam
June 2025

Implementing a Test Generation Service For Flutter Framework

APPROVED BY:

Committee name here

Committee name here

Committee name here

Committee name here

Committee name here

THESIS COMMITTEE

Acknowledgments

It is with profound gratitude and sincere appreciation that I extend my heartfelt thanks to Dr. Tran Thanh Tung for his unwavering support and exceptional professional guidance throughout the course of this thesis. His dedication, insightful feedback, and encouragement provided me with the optimal conditions to carry out and complete this research successfully. Dr. Tran Thanh Tung's invaluable knowledge and expertise have been a constant source of motivation and inspiration, significantly contributing to my learning process and academic growth.

I also want to express my thanks to all professors and lecturers who have followed and instructed me throughout my university journey. Their expertise and experience have enlightened and sharpened my skills to confidently enter the industry.

Lastly, I sincerely thank the thesis evaluation committee for their valuable time reviewing and assessing this thesis.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Abstract

Software testing is indispensable for ensuring the reliability and correctness of any software product before deployment. Despite its importance, developers often find writing unit tests and integration tests tedious and time-consuming. This is not due to the complexity of the process but to the cognitive effort required to work retrospectively, evaluating and validating code logic that has already been implemented without being bias from the logic of the source code.

This thesis introduces an innovative approach leveraging the capabilities of Artificial Intelligence (AI) called "Test Genie", which will alleviate developers' workloads by automating the generation of test cases. By offloading the task of test generation to an AI-driven system, developers can concentrate entirely on writing robust and functional source code. The proposed solution employs the Retrieval-Augmented Generation (RAG) technique to enhance the quality and relevance of the generated test cases, ensuring that the results align with the intended behavior of the code.

To further validate the practicality of the system, the service incorporates an embedded Software Development Kit (SDK) for the supported platform, with the initial implementation focused on the Flutter framework. This integration ensures that the AI-generated test files adhere to the platform's standards and are executable without manual intervention.

The results of this research aim to demonstrate how AI can transform the software testing process, reducing developer effort, improving testing efficiency, and fostering higher-quality code in modern software development.

Chapter 1
INTRODUCTION

1.1 Background

As software systems become increasingly complex, the demand for rigorous software testing has grown significantly. Modern applications often integrate multiple components, rely on distributed architectures, and interact with various external systems, making them more vulnerable to errors. According to a study by Capgemini (2021), the average cost of software failures has risen by 15% annually [1], underscoring the need for comprehensive testing to ensure reliability. Furthermore, the adoption of agile and DevOps methodologies has accelerated development cycles, necessitating continuous testing to maintain quality. The World Quality Report (2022) highlights that 78% of organizations have increased their investment in testing tools and resources over the past five years [1], reflecting the growing recognition of testing as a critical component of software development. Due to high demand in software testing, the market value of digital assurance also get higher. The average annual salary of Quality assurance tester have increased, from 60,000$ in 2015 to 82,000$ in 2024 [2].

1.2 Problem Statement

The rapid evolution of technology has led to the proliferation of programming languages and development frameworks, each with unique features and ecosystems. While this diversity offers developers powerful tools and improved syntax to enhance productivity, it also introduces significant challenges in the testing process. Developers must familiarize themselves with different testing languages, frameworks, and techniques for each platform, which can be both time-consuming and error-prone.

Although languages and frameworks are getting better in both syntax and community support, the testing process also getting trickier. Writing comprehensive unit and integration tests often requires developers to think "backtrackingly," reconstructing potential use cases and edge cases after implementing the functionality. A human can overlooking critical edge cases that might cause a costly consequence. According to CISQ, poor software cost the U.S. economy $2.08 trillion in 2020 alone [3].

To address these challenges, this thesis proposes the integration of an AI-driven Test Generation Service named Test Genie. By leveraging Large Language Models (LLMs), this service automates the creation of test cases, significantly reducing the burden on developers. Automating this process not only optimizes resource allocation but also minimizes the potential for human error, ensuring a more thorough and systematic approach to software testing.

1.3 Scope and Objectives

Initially, this thesis will only focus on one single framework: Flutter - a cross-platfrom framework that can build the product for many platform from one source code. Although Flutter is considered a new framework but the support community and the usage of this framework is increasing every year. This framework also support a testing module, enable users to develop different testing packages and techniques. The research will assess the feasibility of AI in test cases generation by using Langchain library to integrate API of LLM models. By using multiple LLM models, the thesis aim to present a suitable methodology that could provide support to reduce QA testers and developers's workload and effectively cover edge cases that human often miss.

To successfully implement this service, three primary objectives must be achieved. First, the AI must demonstrate the capability to analyze the business model and functional requirements directly from the project source code. This requires understanding the logical structure and intent of the application. Second, the AI must leverage an effective test generator model capable of producing test cases that align with the platform's standards while maintaining relevance to the identified business logic. Third, the generated code must be thoroughly validated to ensure its correctness and compatibility within the Flutter ecosystem. By meeting these requirements, the proposed service aims to establish a reliable and efficient solution for automating test case generation.

In this thesis, we will work on three components:

- Business Logic Analyzer module (BLA)
- AI-integrated test generation module
- AI test validation module

Each component will share the same tech stack:

- Python [6]: This is a popular high-level language that used widely by AI developers. Its simple syntax and wide range of supportive library help developers effectively implement complex system with minimal syntax.
- Python-Flask [4]: This is a micro web framework for Python. It is lightweight and easy to use, making it suitable for building small to medium-sized web applications.
- Python-Langchain [5]: Langchain is a framework for developing applications powered by Large Language Models (LLMs). This is an open-source framework and effectively utilize API provided by LLMs service provider as well as self-hosted LLMs.

1.4 Structure of thesis

This thesis consist of six chapters:

- **Chapter 1. Introduction:** Introduce the background story, how I identify the problem as well as the scope and objectives of this research. This chapter also lightly introduce the proposed solution of the stated problem.
- **Chapter 2. Liturature review/Related work:** This chapter focus on the related work that contributed to the thesis.
- **Chapter 3. Methodology:** Presenting the methodology behind the project, including the component of the system, method implemented for each module and the plan to validate the generated test from AI.

- **Chapter 4. Implement and results:** This chapter summarize the design and implementations of the system as well as the result of this research.
- **Chapter 5. Discussion and evaluation:** In this chapter, we will evaluate the result of this system.
- **Chapter 6. Conclusion and future work:** This chapter will conclude the research of this thesis, as well as the plan of development in the future.

Chapter 2
LITURATURE REVIEW/RELATED WORK

2.1 Unit test generator

Evolution of Test Generation Approaches. Software testing has evolved significantly over the decades, transitioning from purely manual testing to increasingly automated approaches. The earliest automated test generation methods emerged in the 1970s with simple input-output validation techniques [11]. By the 1990s, more sophisticated approaches began to appear, including symbolic execution and model-based testing. The 2000s saw the rise of search-based software testing (SBST), which applies metaheuristic search techniques to generate test cases that satisfy specific coverage criteria [12]. In parallel, constraint-based testing evolved to leverage constraint solvers for generating test inputs that exercise specific code paths. Random testing, despite its simplicity, has remained relevant due to its ability to discover unexpected failures with minimal assumptions about the system under test.

These traditional approaches have dominated the automated testing landscape until recently, when the emergence of advanced artificial intelligence techniques, particularly Large Language Models (LLMs), introduced a paradigm shift in how test generation can be approached. Unlike previous methods that relied on explicit algorithms or search strategies, LLM-based approaches leverage patterns learned from vast corpora of code to generate tests that more closely resemble those written by human developers.

LLMs approach compared to formulated approach. To accurately give test case with correct syntax, I have researched some techniques that can handle different frameworks with just one centrialized system. There is a research that compares the performance of some common approaches including search-based, constraint-based and random-based. Tests generated by these methods frequently lack meaningful structure or descriptive naming conventions, making them difficult for developers to interpret and modify [7]. This limitation can hinder their practical usability, particularly in dynamic and iterative development environments.

In contrast, test case generation using Large Language Models (LLMs) offers a more intuitive and human-aligned approach [7]. LLMs, trained on vast amounts of programming-related data, possess the capability to generate test cases that not only adhere to syntactical correctness but also align closely with human developers' intentions and coding practices. This alignment results in unit tests that are more readable, contextually relevant, and easier to understand. Developers can quickly adjust and refine these tests as needed, enhancing their utility in real-world scenarios.

Moreover, the flexibility of LLMs enables them to adapt seamlessly to various programming languages and frameworks, providing a centralized solution for diverse development ecosystems. While traditional approaches may produce marginally higher percentages of technically correct test cases, they often lack the usability and adaptability that LLM-based methods provide. As a result, services leveraging LLMs for test generation consistently receive more favorable user feedback due to their focus on developer experience, ease of use, and alignment with real-world development workflows.

Quantitative Analysis of Test Generation Approaches. Recent empirical studies have provided quantitative evidence comparing traditional and LLM-based test generation approaches across multiple dimensions. According to a comprehensive benchmark study by Watson et al. [13], test cases generated by LLMs achieved an average of 76% code coverage compared to 82% from specialized constraint-based tools. However, when measuring test suite maintainability using the Test Maintainability Index (TMI), LLM-generated tests scored significantly higher with an average score of 68 compared to 42 for traditional methods. This highlights a fundamental trade-off between technical perfection and practical usability.

Table 2.1 provides a comparative analysis of different test generation approaches based on several key metrics, synthesized from multiple research studies [7, 13, 14].

Table 2.1: Comparison of Test Generation Approaches

Metric	**Search-based**	**Constraint-based**	**Random-based**	**LLM-based**
Code Coverage	High (75-85%)	Very High (80-90%)	Medium (50-65%)	High (70-80%)
Test Readability	Low	Medium	Very Low	Very High
Naming Conventions	Poor	Moderate	Poor	Good to Excellent
Framework Adaptability	Low	Low	Medium	High
Edge Case Detection	Medium	High	Medium-High	Medium
Maintenance Effort	High	Medium	High	Low
Generation Speed	Fast	Slow	Very Fast	Medium
Developer Satisfaction	Low	Medium	Low	High

The data reveals that while traditional methods may excel in specific technical metrics like edge case detection (particularly constraint-based approaches), LLM-based methods offer a more balanced profile with particular strengths in human-centric metrics such as readability and maintainability. This balance makes LLM-based approaches especially suitable for industrial applications where developer productivity and code maintainability are primary concerns.

Disadvantages of LLMs. One of the most significant challenges is their propensity to generate hallucinations, where the model produces incorrect or fabricated outputs that lack grounding in factual data. This issue is particularly critical in tasks requiring precision, such as author attribution. For instance, research introducing the Simple Hallucination Index (SHI) revealed that even advanced LLMs like Mixtral 8x7B, LLaMA-2-13B, and Gemma-7B suffered from hallucinations, with Mixtral 8x7B achieving an SHI as high as 0.87 for certain datasets [8]. These hallucinations undermine the reliability and trustworthiness of LLMs, especially in contexts where factual accuracy is crucial.

Another drawback of LLMs is their lack of transparency in decision-making. These models function as black boxes, providing little insight into the reasoning behind their outputs [8]. This opacity complicates the debugging process and limits the ability to verify results, which is particularly problematic in applications requiring a high degree of explainability. Additionally, LLMs are highly dependent on the quality and diversity of their training data. Biases or inaccuracies present in the data can result in outputs that reinforce those biases or produce flawed results. Moreover, while these models excel at generating output based on their training corpus, they often struggle to generalize effectively when faced with novel or unseen cases.

LLM Limitations in Testing Contexts. While general LLM limitations are well-documented, their specific impact on test generation presents unique challenges. Test generation requires a deep understanding of program semantics, expected behaviors, and framework-specific testing conventions. Wang et al. [15] identified several testing-specific limitations in their comprehensive evaluation of LLM-based testing tools:

First, LLMs frequently generate **syntactically valid but semantically incorrect** tests, particularly when dealing with complex object relationships or state-dependent behaviors. In their study, approximately 32% of generated tests contained semantic errors despite being syntactically correct. Second, LLMs demonstrate **inconsistent mocking behavior**, struggling to correctly identify which components should be mocked in unit tests and how to implement those mocks appropriately. Third, there exists a **framework understanding gap**, where LLMs may mix testing conventions from different frameworks or misapply testing patterns. For Flutter specifically, Li and Zhang [16] found that LLMs often confused widget testing and integration testing approaches, leading to inefficient or incorrect test implementations.

These limitations highlight the need for specialized approaches when applying LLMs to test generation tasks. Solutions proposed in recent literature include fine-tuning models on framework-specific testing examples, implementing post-processing validation steps, and incorporating human feedback loops to refine and correct generated tests [17].

2.2 Understanding Business Logic

The concept of Business Logic. An industry's business logic can be seen as a description of a number of basic conditions or circumstances that make up important starting points for understanding an established business and its conditions for change [9]. It encodes the real-world policies, procedures, and processes that govern how data is created, managed, and manipulated in a way that aligns with the objectives of the organization. Business logic acts as the foundation for decision-making and operational tasks, ensuring that the software performs actions that mirror the intended business behavior. This could involve calculating prices, validating transactions, or managing inventory, all based on predefined rules and conditions derived from the organization's requirements.

Business logic serves as the intellectual layer of an application, translating business needs into functional processes that can be executed by the software. It defines the constraints, relationships, and actions that underpin the flow of data within the system, ensuring that each operation adheres to the intended policies and delivers accurate results. The clarity and accuracy of business logic are essential for maintaining the reliability of software systems, as it directly influences how well the software aligns with the real-world scenarios it is designed to address. By formalizing business rules into structured logic, it enables organizations to automate and scale their operations effectively while minimizing the risk of errors and inconsistencies.

Taxonomies of Business Rules. Business rules can be categorized into several distinct types, each serving different purposes in the overall business logic architecture. Researchers have proposed various taxonomies to classify these rules. One widely accepted classification by von Halle [18] divides business rules into five primary categories:

Definitions form the foundational terms and concepts within a business domain, establishing a common vocabulary. **Facts** express relationships between definitions, capturing the static structure of business information. **Constraints** represent business rules that restrict actions or states, enforcing boundaries on operations. **Derivations** are rules that calculate values or derive new facts from existing data, often implementing business formulas or algorithms. **Action enablers** trigger specific actions when certain conditions are met, representing the dynamic behavior of the system.

Morgan [19] offers an alternative categorization focusing on implementation aspects: **Computational rules** perform calculations following specific algorithms; **Constraint rules** validate data against defined conditions; **Inference rules** draw conclusions based on existing facts; **Process control rules** govern workflow sequences. Understanding these taxonomies is crucial for effective extraction and representation of business logic in test generation systems, as different rule types require different testing approaches and validation strategies.

Existing method. The extraction of business logic from source code has been a long-standing challenge, especially in the context of legacy systems. Traditionally, reverse engineering techniques have been employed to bridge the gap between low-level implementation details and high-level conceptual models of software systems. Tools such as SOFT-REDOC have been developed to support this process, particularly for legacy COBOL programs [9]. These tools rely on program stripping, wherein non-essential code is eliminated to focus on the logic that directly affects specific business outcomes. This involves identifying critical variables and their assignments, conditions, and dependencies to reconstruct the underlying business rules.

Evolution of Business Logic Extraction. Business logic extraction techniques have evolved considerably over time, adapting to changing programming paradigms and technologies. The earliest methods focused on manual code review and documentation, requiring domain experts to manually analyze source code and extract business rules [20]. This approach, while thorough, proved time-consuming and inconsistent. The 1990s saw the emergence of the first automated tools for COBOL and other legacy languages, primarily utilizing static analysis techniques to identify data manipulation patterns [9].

As object-oriented programming became dominant, new methods emerged to handle encapsulated business logic. Fisher et al. [21] developed techniques specifically targeting object-oriented systems, focusing on method interactions and inheritance structures to identify business rules. Their approach achieved a 65% accuracy rate in correctly identifying business rules across multiple commercial Java applications. The mid-2000s brought dynamic analysis approaches that observed system behavior during execution to infer business rules, particularly effective for complex event-driven systems [22].

Recent advances incorporate machine learning and natural language processing techniques. Wong and Menzies [23] demonstrated that supervised learning models could identify business logic components with 78% accuracy after training on labeled code samples. Their approach particularly excelled at distinguishing between technical infrastructure code and actual business logic. Most recently, hybrid approaches combining static analysis, dynamic execution traces, and machine learning have shown promise, with Chen et al. [24] reporting 82% precision in business rule extraction across multiple programming languages.

Challenges with Existing Approaches. The reliance on human analysts to interpret outputs and dependencies makes the process time-consuming and error-prone [9]. Furthermore, legacy programs often involve convoluted logic and scattered assignments, making it difficult to reconstruct business rules with precision. In cases where variable names and data structures lack descriptive clarity, analysts may struggle to comprehend the program's intent, leading to incomplete or inaccurate extraction of business logic. These limitations highlight the need for more automated and scalable approaches to understanding business logic in modern and legacy systems.

Modern software architectures introduce additional complexities for business logic extraction. Microservice architectures distribute business logic across multiple services, making holistic analysis challenging. According to Rodriguez et al. [25], business rules in microservice architectures are 47% more likely to be inconsistently implemented compared to monolithic applications, primarily due to their distributed nature. Similarly, event-driven systems encapsulate business logic within event handlers and subscribers, requiring specialized extraction techniques [26].

Framework-specific challenges also exist, particularly for UI-centric frameworks like Flutter. Nguyen and Kim [27] found that Flutter applications frequently embed business logic within UI components, with an average of 28% of business rules implemented directly within widget classes rather than in dedicated business logic layers. This intertwining of presentation and logic complicates extraction efforts and increases the risk of missed or misunderstood rules.

2.3 Test Quality Assessment

Metrics for Test Quality Evaluation. Evaluating the quality of generated test cases is essential for determining their effectiveness and practical utility. Traditional test quality metrics focus primarily on coverage measurements, with code coverage being the most widely used. However, research by Inozemtseva and Holmes [28] demonstrated that high coverage does not necessarily correlate with test effectiveness in detecting faults. This finding has prompted researchers and practitioners to develop more comprehensive quality metrics that consider multiple dimensions of test effectiveness.

Coverage metrics remain valuable but insufficient indicators of test quality. Line coverage, branch coverage, and path coverage provide increasingly detailed insights into which portions of code are exercised by tests, with path coverage offering the most thorough assessment at the cost of computational complexity. **Mutation testing** represents a more robust approach to evaluating test effectiveness by introducing artificial faults (mutations) into the code and measuring how many mutations are detected by the test suite [29]. A high mutation score indicates tests that are sensitive to changes in program behavior, suggesting better fault detection capability.

Beyond technical effectiveness, **maintainability metrics** evaluate how easily tests can be understood and modified. Metrics such as cyclomatic complexity, test size, assertion density, and comment ratio contribute to an overall test maintainability index [30]. Studies by Bavota et al. [31] found that more maintainable tests are 42% more likely to be regularly updated when the code they test changes, highlighting the practical importance of these metrics.

Comparing Generated Tests to Human-Written Tests. Empirical studies comparing AI-generated tests with human-written tests reveal interesting patterns across various quality dimensions. Tufano et al. [32] conducted a blind evaluation where professional developers were asked to review both human-written and AI-generated tests without knowing their origin. Their findings revealed that AI-generated tests achieved comparable technical quality but differed in stylistic elements.

AI-generated tests demonstrated strengths in systematic coverage of edge cases, with 28% more boundary conditions tested on average compared to human-written tests. However, they scored lower on contextual understanding, with human reviewers noting that 34% of AI-generated tests included irrelevant assertions or tested aspects that weren't meaningful to the application domain. Most notably, human-written tests excelled in testing domain-specific behavior that required contextual knowledge not explicitly present in the implementation code [32].

The gap between human and AI test generation has narrowed significantly with recent LLM-based approaches. In a follow-up study using more advanced LLMs, Watson et al. [13] found that professional developers could correctly identify the origin of tests (human vs. AI) only 58% of the time, barely better than random chance. This suggests that modern AI approaches are producing tests increasingly indistinguishable from human-written ones in terms of style and structure, though gaps in domain understanding persist.

2.4 Framework-Specific Testing Challenges

Flutter Testing Ecosystem. The Flutter framework presents unique testing challenges and opportunities due to its cross-platform nature and widget-based architecture. Flutter's testing ecosystem encompasses three primary testing levels: unit testing for individual functions and classes, widget testing for UI components, and integration testing for end-to-end application behavior [33]. Each level requires different testing approaches and introduces distinct challenges for automated test generation.

Unit testing in Flutter follows standard Dart testing conventions but includes additional complexities when testing code that interacts with Flutter's widget system or platform channels. According to Shah et al. [34], the most common challenge in Flutter unit testing is properly mocking dependencies, particularly those that interact with the Flutter framework or platform-specific code. Their analysis of open-source Flutter projects found that 64% of unit test failures were related to improper mocking or dependency isolation.

Widget testing represents a middle ground between unit and integration testing, focusing on testing UI components in isolation. Flutter's widget testing framework provides tools for rendering widgets, simulating user interactions, and verifying expected UI behavior. However, Zhao and Li [35] identified several challenges specific to widget testing, including handling asynchronous UI updates, managing widget lifecycles, and testing complex widget hierarchies. Their study found that widget tests written by novice Flutter developers had a 47% higher failure rate than those written by experienced developers, highlighting the steep learning curve associated with effective widget testing.

Cross-Platform Testing Considerations. Flutter's promise of a single codebase for multiple platforms introduces additional testing considerations. While the core application logic may be shared, platform-specific behaviors, interactions, and appearances often require targeted testing approaches. Research by Martinez and Leiva [36] found that 38% of Flutter application bugs were platform-specific despite the shared codebase, with iOS-specific issues being 1.6 times more common than Android-specific issues in the studied applications.

Testing platform-specific features presents a particular challenge for automated test generation. Kim et al. [37] evaluated several automated testing tools for Flutter and found that none could effectively generate tests for platform channel implementations or platform-specific UI adjustments without significant human guidance. Their proposed solution involved platform-aware test generation that incorporated platform-specific expectations and behaviors into the generated tests.

These framework-specific challenges highlight the need for specialized approaches when generating tests for Flutter applications. Effective test generation must account for Flutter's unique widget lifecycle, asynchronous programming model, and cross-platform considerations to produce meaningful and reliable tests.

Chapter 3

METHODOLOGY

3.1 Overview

The methodology chapter provides a comprehensive overview of the approach taken in this research. It outlines the key components of the system, including the Business Logic Analyzer module (BLA), the AI-integrated test generation module, and the AI test validation module. Each component is designed to work seamlessly together, leveraging Python, Flask, and Langchain to create an efficient and effective solution for automating test case generation. The chapter also discusses the methods implemented for each module and the plan to validate the generated tests from AI, ensuring that the proposed solution meets its objectives and addresses the identified challenges in software testing.

3.2 User requirement analysis

Understanding user requirements is a critical step in ensuring that the proposed system aligns with the needs and expectations of its target audience. This phase involves identifying and analyzing the specific functionalities, constraints, and preferences that users demand from the system. A thorough understanding of user requirements not only guides the development process but also ensures the system delivers value by addressing real-world challenges effectively. This section outlines the key user requirements identified for the proposed test generation service.

Req.ID	Requirement Name	Detailed Description	Type
001	Read project's source code	Users can send all project's source code at once via web-based Git repositories (e.g github, gitlab)	Functional requirement
002	Download/copy unit test/integration test	Users can download tests files or copy the file's content.	Functional requirement
003	Interactive business logic analyzation (Human-inner-loop)	Users can help AI correct the result of BLA process	Functional requirement
004	Performance	The system should generate test cases within a reasonable time frame, ideally under 5 minutes for a medium-sized project (e.g., 10,000 lines of code).	Non-functional requirement
005	Test file correctly reflect the given business model	The system should be able to generate test cases accurately reflect the business logic embedded in the source code.	Non-functional requirement
006	Validate generated test	A validation mechanism must be included to the system to ensure the syntax and logic is runnable	Non-functional requirement

Table 3.1: User requirements

3.2.1 Ability to send project's source code

The Test Genie system requires users to submit their project's source code via web-based Git repositories (e.g., GitHub, GitLab) rather than traditional methods like ZIP files. This design is intentional and aligns with modern development workflows since most modern projects have an online git repository. The biggest advantage is that this method will optimize unneeded directory that will be added to gitignore by users. Some modern framework use library that is sometimes heavy and not necessary during Business Logic Analyze process. Not adding these files will optimize the workloads of system much better.

User flow. Users will input the Git repository link via the User Interface (UI) and select the desired branch for analysis. If the system encounters access issues or cannot connect to the repository (e.g., internal Git systems), it will respond with an error message, prompting the user to resolve the issue.

System flow. After receiving the Git link and branch information, the system will clone the repository. Using predefined tokens or configuration files (e.g., pubspec.yaml for Flutter), the system will identify the framework and dependencies used in the project. Based on this information, the system will apply the most suitable strategy to analyze the source code and generate test cases.

3.2.2 Give user output

The output of the system is a full test file content that can be integrate into their existing workflows. The output is delivered through a live chat downloadable UI, ensuring a seamless and interactive experience for users.

Output format. Currently, this system only supports the Flutter framework, which has a built-in testing system. The system generates test files with the naming convention *"filename_test.dart"*, where the filename corresponds to the specific module or functionality being tested. This naming convention ensures that the test files are easily identifiable and organized within the project structure. The content of the test files is tailored to match the testing requirements requested by the user, including unit tests, integration tests, or widget tests, depending on the analysis of the source code. By adhering to Flutter's testing standards, the generated files are immediately compatible with the framework, allowing developers to run the tests without additional configuration. This approach ensures that the output is not only functional but also aligns with best practices for Flutter development.

Live chat interface. Users receive the generated test files through a live chat interface embedded in the system's UI. This interface provides a real-time, interactive experience, enabling users to communicate with the system as it generates and refines test cases. For example, if the user identifies an issue with the generated tests (e.g., incorrect logic, missing edge cases, or mismatched parameters), they can provide feedback directly through the chat. The system will then process this feedback and adjust the test cases accordingly. This two-way communication ensures that the final output meets the user's expectations and aligns with the project's requirements. Additionally, the live chat interface can provide explanations or suggestions for improving the tests, making it a valuable tool for both novice and experienced developers. This interactive approach enhances user satisfaction and ensures that the generated tests are accurate and relevant.

Downloadable Files. Instead of requiring users to manually create and organize test files, the system allows users to download the generated files directly and save them in the /tests/ folder of their Flutter project. This feature eliminates the need for manual file creation and ensures that the tests are placed in the correct directory, adhering to Flutter's project structure. The files are packaged in a format that is ready to be integrated into the user's project, requiring minimal manual intervention. This seamless integration reduces the risk of errors and saves developers' valuable time. Furthermore, the system ensures that the downloaded files are compatible with version control systems like Git, allowing users to immediately commit the tests to their repository. This feature is particularly useful for teams working in collaborative environments, as it streamlines the process of adding tests to the codebase.

Easy to adjust. Although the system is embedded with a validator to ensure that the generated tests are syntactically correct and runnable, it recognizes that real-world scenarios may require adjustments. For instance, the system might generate tests based on default parameters or assumptions that do not fully align with the user's specific use cases. In such situations, users can easily adjust the test parameters to better fit their requirements. The system provides clear and well-structured test files, making it straightforward for developers to modify variables, inputs, or assertions as needed. This flexibility ensures that the generated tests remain useful even in complex or unique scenarios. By combining automated test generation with the ability to manually refine the results, the system strikes a balance between efficiency and adaptability, catering to a wide range of development needs.

3.2.3 Interactive Business Logic Analyzating process

The Business Logic Analyzing (BLA) process plays a crucial role in ensuring that the system accurately interprets and applies business logic. If the output of this process is incorrect, it can lead to downstream malfunctions and errors, which can be costly and time-consuming to resolve. To address this, the system incorporates an interactive BLA process that allows users to collaborate with the AI to improve analysis results.

User interface. The interface for this process is designed to be intuitive and user-friendly, enabling users to interact with a visual representation of the project's modules, classes, and functions in the form of a graph. This graphical layout provides a clear overview of how different components of the application are interconnected and functioned. Users can inspect the analysis results by interacting with this graph, allowing them to identify potential issues or discrepancies in the current output.

One key feature of this interface is its ability to be manipulated by users. Through inspection, users can help guide the AI by highlighting specific areas of interest, providing context, or pointing out errors in the analysis. This interactive capability allows for a more precise and accurate understanding of how the business logic is being applied within the system.

Sytem flow. Once the project's source code has been submitted to the system, it undergoes an initial analysis phase that maps out the relationships between classes, modules, and functions. The system uses this information to generate a detailed breakdown of the project's structure and flow. After the analysis is complete, users receive access to a project insight webview that provides a comprehensive visual representation of how these components interact with each other.

This webview not only displays the flow of the project but also highlights any potential issues or areas where the business logic may require adjustment. The system ensures that this visualization is clear and concise, making it easy for users to understand and address any discrepancies in the analysis.

3.2.4 Optimize performance

The input of this system is the user's source code of the project they needed to generate. A study show that the average number lines of code (LOC) of a project with 90 functions will have 90,000 lines of codes [10]. From AI perspective, that is an enormous amount of input tokens. To handle these input lighter, these inputs will be split into blocks of component to analyze.

Splitting strategy. In this system, relational database will be used to store project's source code. Each component will contain the input, output, related component information and the predicted business logic of that component. This structured approach allows for efficient handling and analysis of large inputs while maintaining clarity and organization.

Quering component. The graphical webview that was introduced above will be contruct by query the connection of these component.

Performance overall. By organizing the input into blocks of component and using efficient querying mechanisms, the system optimizes its ability to handle large-scale projects without compromising performance. The use of a relational database ensures that data retrieval is both organized and efficient, reducing the likelihood of bottlenecks during analysis.

This approach not only enhances the system's capacity to process extensive codebases but also improves overall efficiency by minimizing redundant data storage and retrieval processes.

3.2.5 Good test file generation - Quality control

To ensure high-quality test file generation while maintaining the abstraction of the LLM model, this thesis adopts the Retrieval-Augmented Generation (RAG) technique. This approach involves embedding relevant project framework documents (currently focused on Flutter) and providing them as input to the model through structured prompts. By augmenting the model with specific, context-rich information, the system can generate test cases that better align with the framework's requirements and coding standards.

Provided documents. The documents supplied to the LLM are carefully selected to include essential information related to testing syntax, techniques, and best practices for the Flutter framework. These resources guide the model in generating syntactically correct and framework-compliant test cases.

User-side documents. Users have the option to provide supplementary documents and sample test files from their projects. This customization allows the system to learn and adhere to the specific naming conventions, organizational structures, and testing styles already established within the project.

3.2.6 Test validation

In this thesis, the validation scope focuses on ensuring that the generated test files are runnable within the intended development environment. Rather than validating the correctness of test outcomes or the business logic they cover, the emphasis is placed on generating test files that can be successfully executed without syntax or framework-related errors. To achieve this, a Software Development Kit (SDK) is embedded for each supported framework, with the initial implementation targeting the Flutter framework. This SDK integration ensures compatibility with the framework's testing infrastructure, allowing the generated tests to be seamlessly executed as part of the development workflow. By embedding the SDK, the system can identify and address potential issues during the test generation process, such as missing dependencies or incorrect file structures, thereby increasing the reliability of the output. While the current scope does not extend to evaluating the correctness of test assertions or coverage, this foundational validation approach ensures that developers receive test files that are syntactically correct, executable, and immediately ready for further refinement or deployment within their projects. Future enhancements may involve integrating more advanced validation techniques, such as logic verification

3.3 System Design

3.3.1 Core Design Philosophy

The fundamental challenge in AI-based test generation is source code bias, where the AI model's exposure to implementation details leads to tests that merely replicate code behavior rather than validating business requirements. Test Genie addresses this challenge through a novel architectural approach based on modular analysis and isolation of concerns.

Source code bias occurs when an AI model, given complete access to implementation details, generates tests that are essentially tautological—they validate that the code does what the code does, rather than what it should do according to business logic. This issue undermines the purpose of testing as an independent verification mechanism. Traditional approaches either restrict AI access to implementation details (limiting effectiveness) or accept this bias as inevitable.

Test Genie's approach is fundamentally different. By decomposing source code into discrete, semantically meaningful blocks and analyzing them in isolation, the system creates a separation between implementation and testing concerns. Each block represents a distinct functional unit with clear inputs, outputs, and business logic implications. This decomposition allows the system to:

- Generate accurate predictions about each block's purpose without being overwhelmed by the complexity of the entire codebase
- Focus on functional intent rather than implementation details
- Isolate business logic from technical implementation
- Enable effective human-in-the-loop correction at a manageable granularity

This modular approach offers significant advantages over traditional methods. While conventional test generation might produce tests that trivially pass because they mirror the implementation logic, Test Genie's block-based analysis encourages tests that validate the expected behavior of each component according to its business purpose. Figure 3.1 visualizes this architectural philosophy, showing how decomposition into blocks enables more effective analysis and test generation.

3.3.2 System Architecture

The Test Genie system implements a three-tier architecture consisting of the **User Interface (UI)**, the **Request Handler** middleware, and the **Application Service (Backend)** layer. This architecture facilitates clear separation of concerns, enabling each component to fulfill its specific role in addressing the source code bias problem. Figure 3.1 illustrates the overall module design of the system.

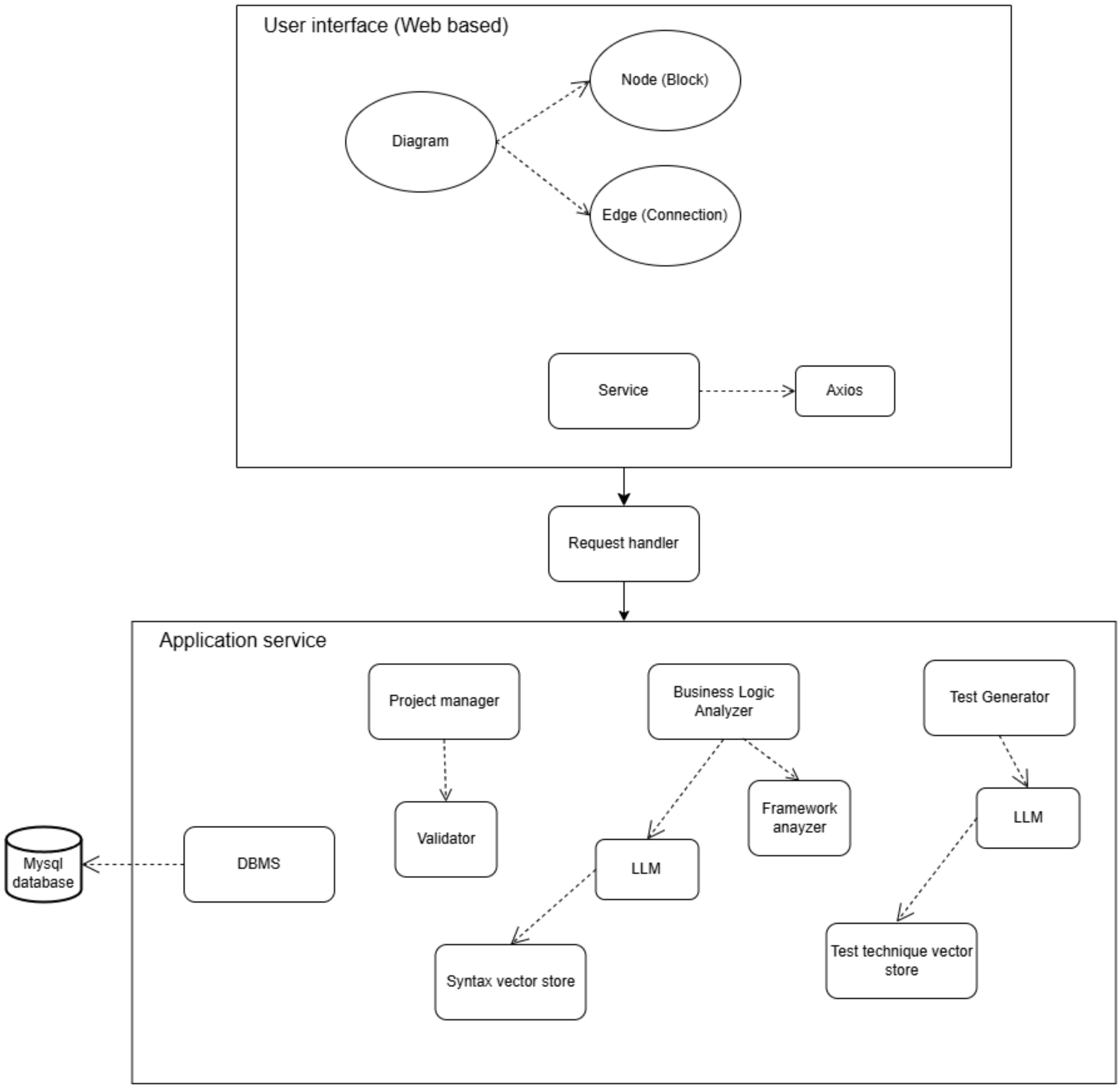

Figure 3.1: Test Genie's overall module design

User Interface (UI) Layer

The User Interface layer serves as the primary interaction point between users and the Test Genie system. Built as a web-based application, it combines visualization capabilities with interactive elements designed specifically to facilitate human-in-the-loop participation in the analysis and test generation process. Key components include:

- **Interactive Graph Visualization**: Renders the source code structure as a navigable graph of blocks and connections, allowing users to visually comprehend complex project architectures. The visualization employs force-directed graph algorithms to optimally arrange components based on their interconnections.

- **Block Inspector**: Provides detailed views of individual code blocks, including:
  - **Source Code View**: Syntax-highlighted representation of the block's actual implementation
  - **Prediction Editor**: Interactive interface for viewing and refining the AI-generated predictions about the block's business purpose

  - **Test Preview**: Live preview of generated test cases with syntax highlighting

- **Human-in-the-Loop Controls**: Interface elements that enable users to:
  - Navigate between related blocks and understand their connections
  - Modify AI-generated predictions when they misinterpret business logic
  - Trigger test regeneration after prediction updates
  - Download or copy generated test files

The UI is designed with a focus on transparency and interpretability, allowing users to understand and influence the AI's reasoning process. This transparency directly addresses the source code bias problem by enabling users to identify and correct cases where the AI has focused too heavily on implementation details rather than business intent. By providing visual representation of blocks and their relationships, the UI helps users mentally separate implementation from specification, reinforcing the system's core philosophy.

Request Handler Layer

The Request Handler layer serves as the communication bridge between the UI and backend services, implementing a RESTful API architecture that processes user interactions and coordinates system responses. This layer implements several key functions:

- **Request Routing and Validation**: Ensures that user requests are well-formed and directed to the appropriate backend services
- **Caching and Performance Optimization**: Maintains a cache of frequently accessed data to minimize redundant computation and database queries
- **Session Management**: Tracks user sessions and maintains contextual information about ongoing analysis tasks
- **Response Formatting**: Transforms backend results into structured data formats suitable for UI presentation

Table 3.2 outlines the primary API endpoints implemented by the Request Handler layer:

Endpoint	Method	Purpose
/createProject	POST	Initialize project analysis by providing Git repository URL
/getDiagram	POST	Retrieve block and connection data for visualization
/getBlockContent	POST	Fetch source code and metadata for a specific block
/getBlockDetail	POST	Retrieve comprehensive information about a block, including predictions and tests
/getBlockPrediction	POST	Retrieve the current business logic prediction for a block
/updateBlockPrediction	POST	Update the business logic prediction for a block based on user input
/generateTest	POST	Generate test cases for a specified block using current predictions

Table 3.2: Request Handler API Endpoints

The Request Handler's implementation of these endpoints ensures that user interactions with the UI efficiently translate to appropriate actions in the backend services. This layer's design is critical to maintaining the system's responsiveness during the iterative process of analysis, prediction refinement, and test generation.

Application Service (Backend) Layer

The Backend layer contains the system's core functionality, implementing the computational processes that analyze source code, generate predictions, and create test cases. This layer comprises several specialized modules working in concert:

Project Manager The Project Manager module serves as the gateway to source code analysis, with responsibilities including:

- **Repository Handling**: Cloning Git repositories, managing local copies, and extracting relevant files
- **Framework Detection**: Identifying the programming framework (currently focusing on Flutter) by analyzing project structure and configuration files
- **Source File Extraction**: Identifying and extracting source files relevant for analysis while filtering out non-essential files
- **Test Environment Setup**: Creating and maintaining isolated environments for test execution and validation
- **Test Execution**: Running generated tests against the embedded framework SDK to validate correctness

The Project Manager implements framework-specific adapters (currently Flutter) that encapsulate knowledge about project structures, file organizations, and testing conventions. This approach allows for future extensibility to additional frameworks while maintaining a consistent interface for other system components.

Business Logic Analyzer (BLA) The Business Logic Analyzer represents the system's analytical core, implementing the block-based decomposition approach central to Test Genie's design philosophy. The BLA performs several critical functions:

- **Source Code Parsing**: Converting raw source files into abstract syntax trees (ASTs) that can be analyzed programmatically
- **Block Identification**: Applying heuristic algorithms to identify semantically meaningful code blocks such as methods, functions, and classes
- **Connection Analysis**: Determining relationships between blocks based on method calls, inheritance, and other dependencies
- **Block Prediction Generation**: Applying AI analysis to each individual block to predict its business purpose

The BLA's block identification process employs a specialized algorithm designed to identify code units that represent discrete functional components with well-defined inputs and outputs. This decomposition is central to addressing source code bias, as it allows the system to analyze each component's intended purpose without being overwhelmed by implementation details of the entire codebase.

Algorithm 1 outlines the block identification process:

Algorithm 1 BlockIdentification(SourceFiles)

Require: $SourceFiles$ is a list of source code files from the project

$Blocks \leftarrow \emptyset$ ▷ Initialize empty block collection
$Connections \leftarrow \emptyset$ ▷ Initialize empty connections collection
for each $file \in SourceFiles$ **do**
  $ast \leftarrow ParseSourceToAST(file)$
  $fileBlocks \leftarrow ExtractBlocksFromAST(ast)$
  for each $block \in fileBlocks$ **do**
    $block.id \leftarrow GenerateUniqueIdentifier()$
    $block.name \leftarrow ExtractBlockName(block)$
    $block.type \leftarrow DetermineBlockType(block)$
    $block.content \leftarrow ExtractSourceCode(block)$
    $block.originalFile \leftarrow file.path$
    $Blocks \leftarrow Blocks \cup \{block\}$
  end for
  $fileConnections \leftarrow IdentifyConnectionsInFile(fileBlocks)$
  $Connections \leftarrow Connections \cup fileConnections$
end for
$crossFileConnections \leftarrow IdentifyCrossFileConnections(Blocks)$
$Connections \leftarrow Connections \cup crossFileConnections$
for each $block \in Blocks$ **do**
  $block.prediction \leftarrow GeneratePrediction(block, Blocks, Connections)$
end for
return $(Blocks, Connections)$

The prediction generation process employs an AI-based approach that combines contextual understanding with code analysis, leveraging language models enhanced with domain-specific knowledge of programming patterns and testing techniques.

Test Generator The Test Generator module transforms business logic predictions into executable test cases tailored to the specific framework (currently Flutter/Dart). Key features include:

- **Context-Aware Test Creation**: Generating tests that validate business requirements rather than implementation details
- **Test Framework Integration**: Producing tests compatible with the target framework's testing infrastructure
- **Dynamic Adjustment**: Adapting test generation based on user feedback and prediction refinements
- **Test Validation**: Verifying that generated tests are syntactically correct and executable

The Test Generator addresses source code bias by focusing on testing the predicted business purpose rather than the implementation details. By generating tests based on predictions about what the code should do (which can be corrected by users if necessary) rather than what the code actually does, the system produces tests that provide genuine validation rather than tautological verification.

Vector Stores The Vector Stores component implements a Retrieval-Augmented Generation (RAG) approach to enhance AI performance with domain-specific knowledge:

- **Syntax Vector Store**: Contains embeddings of programming language syntax, patterns, and idioms
- **Test Technique Vector Store**: Maintains embeddings of testing best practices, patterns, and framework-specific approaches

These vector stores enable the AI components to access relevant domain knowledge during analysis and generation tasks, producing more accurate and contextually appropriate outputs. By embedding knowledge about effective testing techniques, the system encourages tests that verify behavior against requirements rather than implementation.

Database Management System (DBMS) The DBMS provides persistent storage and efficient retrieval mechanisms for the system's data structures:

- **Block Storage**: Maintains records of identified code blocks, their content, and associated metadata
- **Connection Management**: Stores relationships between blocks, forming a navigable graph structure
- **Prediction Tracking**: Records and updates predictions for each block
- **Test Case Storage**: Stores generated test cases and their validation status

The DBMS schema, illustrated in Figure 3.2, supports the block-based decomposition central to the system's approach to mitigating source code bias.

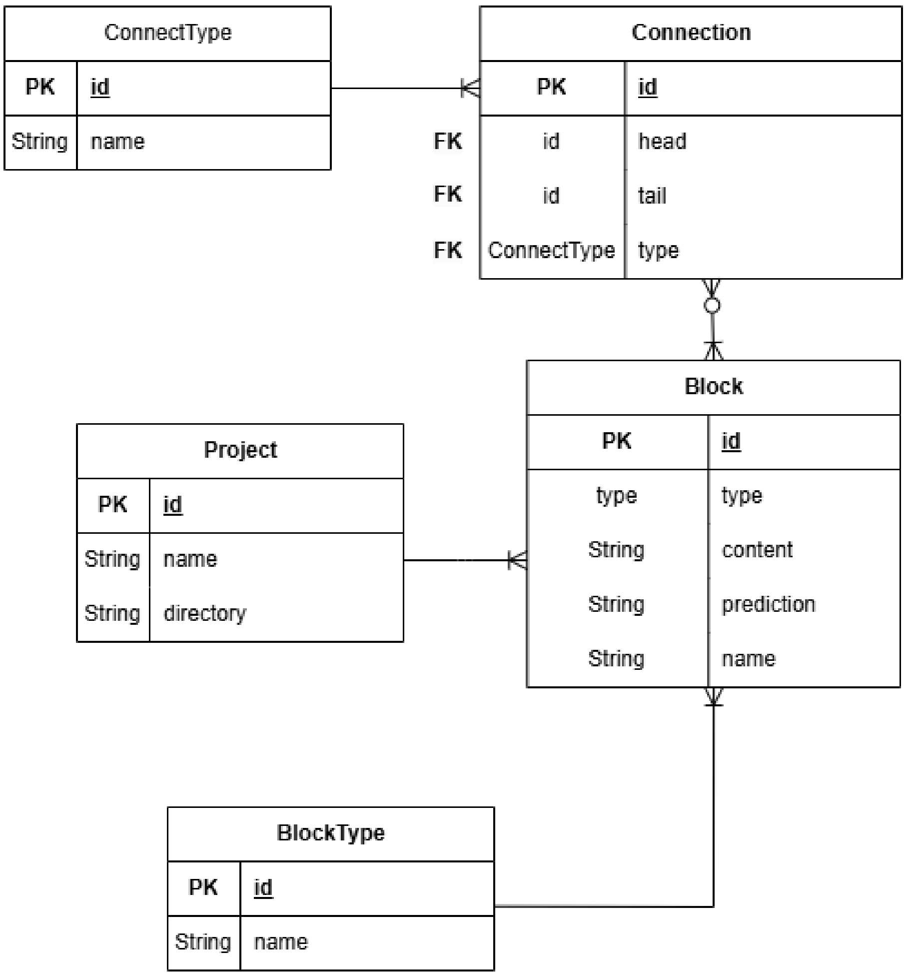

Figure 3.2: Block Relational Database Design

3.3.3 System Workflow

Test Genie's workflow implements a comprehensive process from source code intake to test generation, with specific mechanisms to address source code bias at each stage. The workflow proceeds through several distinct phases:

Project Initialization

1. **Repository Acquisition**: The system clones the user-specified Git repository

2. **Framework Detection**: Project structure is analyzed to identify the programming framework

3. **Configuration Extraction**: Framework-specific configuration files are parsed to understand project organization

Source Code Analysis

1. **File Filtering**: Non-essential files (e.g., assets, configurations) are excluded from analysis
2. **AST Generation**: Source files are parsed into abstract syntax trees
3. **Block Identification**: The BLA algorithm identifies discrete functional blocks
4. **Connection Mapping**: Relationships between blocks are discovered and cataloged

Business Logic Analysis

1. **Block-Level Analysis**: Each block is analyzed independently to focus on its specific purpose
2. **Contextual Enrichment**: Block analysis is enhanced with limited information about connected blocks
3. **Prediction Generation**: The system generates predictions about each block's business purpose
4. **Prediction Storage**: Predictions are stored in the database for future reference and refinement

Human-in-the-Loop Refinement

1. **Visualization**: The block structure is visualized in the UI as a navigable graph
2. **Prediction Review**: Users can inspect and refine the AI-generated predictions
3. **Contextual Understanding**: The visualization helps users understand how blocks interact
4. **Feedback Integration**: User corrections are stored and used in subsequent analysis

Test Generation

1. **Block Selection**: Users select specific blocks for test generation
2. **Context Assembly**: The system gathers relevant information about the selected block and its connections
3. **Test Strategy Determination**: Based on block type and prediction, appropriate testing strategies are selected
4. **Test Case Generation**: The system generates test cases focused on validating the predicted business purpose
5. **Test Validation**: Generated tests are checked for syntax correctness and executability

Test Refinement

1. **Test Preview**: Users can preview generated tests in the UI
2. **Error Detection**: The system identifies and reports issues found during validation
3. **Automated Correction**: When possible, the system automatically corrects identified issues
4. **Test Finalization**: Users can download or copy the final test files for integration into their project

Throughout this workflow, the block-based approach systematically mitigates source code bias by:

- Analyzing discrete functional units rather than entire codebases
- Focusing on predicted business purpose rather than implementation details
- Enabling human correction of predictions before test generation
- Generating tests that validate intended behavior rather than actual implementation

This workflow represents a significant advancement over traditional approaches that either accept source code bias as inevitable or attempt to avoid it by limiting access to implementation details. By decomposing the codebase into manageable units and enabling human oversight of the analysis process, Test Genie achieves a more effective balance between automation and human expertise.

3.3.4 Technical Implementation Details

Block Decomposition Strategy

Test Genie's approach to block decomposition balances granularity with semantic meaning. The system identifies blocks at several levels of abstraction:

- **File-Level Blocks**: Represent entire source files, capturing overall purpose and structure
- **Class-Level Blocks**: Represent individual classes, encapsulating state and behavior
- **Method-Level Blocks**: Represent methods and functions, capturing specific behaviors
- **Nested Blocks**: Represent significant nested structures like inner classes or complex control flows

This multi-level approach ensures that blocks maintain semantic coherence while remaining manageable for analysis. The system employs framework-specific heuristics to identify meaningful blocks. For Flutter, this includes recognition of widget classes, state classes, and business logic components.

Prediction Generation Approach

The prediction generation process employs a two-phase approach:

1. **Initial Analysis**: Each block is analyzed in isolation to generate a preliminary understanding of its purpose

2. **Contextual Refinement**: Limited information about connected blocks is incorporated to refine the analysis

This approach balances the need for contextual understanding with the goal of avoiding source code bias. By limiting the contextual information to essential relationships rather than implementation details of connected blocks, the system maintains focus on business purpose over implementation mechanisms.

The AI model used for prediction generation is enhanced with domain-specific knowledge through the vector stores, enabling more accurate interpretation of programming patterns and idioms. This enhancement is particularly important for framework-specific constructs, such as Flutter's widget hierarchy and state management patterns.

Test Generation Strategy

Test Genie's test generation strategy focuses on validating the predicted business purpose of each block, implementing a three-phase approach:

1. **Test Planning**: Based on the block's predicted purpose, the system identifies appropriate test scenarios

2. **Test Structuring**: The system creates a framework-compliant test structure with appropriate setup and assertions

3. **Test Validation**: Generated tests are validated for syntax correctness and executability

The test generation process deliberately avoids direct examination of the block's implementation details beyond what is necessary to establish method signatures and input/output types. This approach ensures that tests validate the expected behavior rather than merely confirming the current implementation.

3.3.5 Addressing Source Code Bias Through System Design

Test Genie's architecture directly addresses the challenge of source code bias in AI-based test generation through several key mechanisms:

- **Block-Based Decomposition**: By analyzing discrete functional units rather than entire codebases, the system reduces the complexity of each analysis task and focuses on specific business purposes rather than implementation interactions.

- **Separation of Prediction and Testing**: The system maintains a clear separation between the prediction of business purpose and the generation of tests, allowing each process to focus on its specific objective without contamination.

- **Human-in-the-Loop Refinement**: By enabling users to review and correct AI-generated predictions, the system leverages human expertise to identify cases where the AI has focused too heavily on implementation details rather than business intent.
- **Context-Limited Analysis**: The system deliberately limits the contextual information used in analysis to prevent over-reliance on implementation details of related components.
- **Validation-Focused Testing**: Tests are designed to validate the predicted business purpose rather than to verify the current implementation, ensuring that they provide meaningful quality assurance rather than tautological confirmation.

These design choices represent a fundamental shift from traditional approaches to AI-based test generation. Rather than attempting to generate tests directly from implementation details or avoiding implementation details entirely, Test Genie embraces a middle path that leverages the strengths of both approaches while mitigating their weaknesses.

By decomposing the codebase into manageable blocks, the system makes it feasible to analyze each component's intended purpose without being overwhelmed by implementation complexity. By enabling human oversight of the prediction process, the system leverages human expertise to correct cases where the AI has misinterpreted business intent. And by generating tests based on these refined predictions rather than implementation details, the system produces tests that provide genuine validation rather than merely confirming the current implementation.

This approach offers several advantages over traditional methods:

1. **Improved Test Quality**: Tests validate business requirements rather than implementation details, providing more meaningful quality assurance.
2. **Enhanced Maintainability**: Tests remain valid even as implementations change, as long as the business purpose remains consistent.
3. **Greater Transparency**: The separation of prediction and testing makes the system's reasoning process more interpretable and correctable.
4. **More Efficient Human Oversight**: By decomposing the codebase into manageable blocks, the system makes it feasible for humans to review and correct AI-generated predictions.

Through these mechanisms, Test Genie's architecture effectively addresses the challenge of source code bias in AI-based test generation, offering a more effective and efficient approach to automated testing that balances AI capabilities with human expertise.

3.3.6 Alignment with User Requirements

Test Genie's system design directly addresses the user requirements outlined in Table 3.1:

- **Requirement 001 - Read project's source code**: The Project Manager module implements robust Git repository handling capabilities, supporting major hosting platforms.

- **Requirement 002 - Download/copy unit test/integration test**: The UI provides intuitive mechanisms for downloading or copying generated test files.
- **Requirement 003 - Interactive business logic analyzation**: The block visualization and prediction editing features enable effective human-in-the-loop participation.
- **Requirement 004 - Performance**: The block-based decomposition approach and efficient caching mechanisms ensure reasonable performance even for larger projects.
- **Requirement 005 - Test file correctly reflect business model**: The separation of prediction and testing ensures that tests validate business requirements rather than implementation details.
- **Requirement 006 - Validate generated test**: The embedded SDK integration ensures that generated tests are syntactically correct and executable.

This comprehensive alignment demonstrates the effectiveness of the system design in addressing the identified user needs while simultaneously tackling the challenge of source code bias.

3.3.7 Conclusion

Test Genie's system design represents a significant innovation in automated test generation, addressing the fundamental challenge of source code bias through a novel block-based decomposition approach. By analyzing discrete functional units, enabling human refinement of predictions, and generating tests based on business purpose rather than implementation details, the system produces more meaningful and effective tests than traditional approaches.

The modular architecture, with its clear separation of concerns between UI, middleware, and backend services, provides a robust foundation for future enhancements and extensions. The current implementation, focused on the Flutter framework, demonstrates the viability of this approach while laying groundwork for expansion to additional programming environments.

Through its innovative design and implementation, Test Genie advances the state of the art in automated testing, offering a more effective balance between AI capabilities and human expertise that addresses the limitations of previous approaches.

Chapter 4
IMPLEMENT AND RESULTS

This chapter delves into the implementation of each module inside Test Genie system. Overall, this system consist of three main modules:

- **Project Manager**: This module manages all the projects that are cloned to server. It mostly responsible for file-based activities and running CLI for each project.
- **Business Logic Analyzer**: This module will take various source file from Project Manager and break the source code into smaller pieces (blocks). Then, it will analyze each block and determine what each block does and how it should be tested if possible. A test plan will also be generated for each block and save it to the database.
- **Test Generator**: This module will take the test plan from Business Logic Analyzer and generate a test case for each block. The generated test cases will be saved as files directly in the project source code on server and can be used to run the tests later (validation).

Additionally, this system also have **DBMS** module to control the database but this module will not be explained thoroughly in this chapter.

4.1 Project Manager module

The **ProjectManager** module serves as the core backend functionality for handling projects within the Test Genie system. It provides a robust framework for managing software projects by integrating Git-based repositories, file management, and testing workflows. The module is built around the *Project* class, which encapsulates essential functionalities such as cloning repositories, recognizing project frameworks, and managing project files. Additionally, it features an abstract interface for test creation, validation, and execution, allowing for framework-specific extensions of functionality. For instance, the *Flutter* subclass extends the *Project* class to handle Flutter-specific tasks, including dependency management, ‘pubspec.yaml‘ parsing, and test execution. By modularizing these functionalities, the **ProjectManager** module streamlines project handling and enhances the system's scalability for various software development frameworks.

4.1.1 Module prequisites

This module require the SDK of supported frameworks to be installed standablone in folder *./SDKs* inside the module folder. This design not only allows the module to be easily extended and modifiled to support other frameworks, but also avoid more SDK installation on the server OS. Since the *Project* class (Listing A.1) just mainly control git management and file management, the subclass can freely control how the SDKs are used.

Subclass of Project are required to implement the following methods:

- **create_test**: This method will create the test file in the location that is required by the framework.

- **get_test_content**: This method will return the content of the test file that is created by the **create_test** method. The content of the test file is generated by the Business Logic Analyzer module.

- **run_test**: This method will run the test file that is created by the **create_test** method. The test result will be returned to the caller.

- **validate**: This method will run all the test files in the test directory and return the result. This method is used to validate the test files that are created by the **create_test** method.

- **getListSourceFiles**: This is an important method, which will partly decide how the source code is split into blocks. The starting point file (main file) should be placed on the first position of the list. The list will be used to split the source code into blocks. The list should contain all the source files in the project (relative to the project directory).

4.1.2 Flutter class

The **Flutter** class extends the **Project** class to provide framework-specific support for managing Flutter projects. This class is responsible for handling operations unique to Flutter, such as managing dependencies, running tests, and validating projects. It ensures that the Flutter SDK is installed and properly configured in the *./SDKs/flutter* directory before performing any operations.

Key methods of the **Flutter** class include:

- **_runFlutterCLI**: This method executes commands using the Flutter CLI within the context of the project directory. It supports arguments for various Flutter commands and handles errors if the command fails.

- **_checkSDK**: Ensures that the Flutter SDK is installed and operational by running the `flutter --version` command. If the SDK is not present or misconfigured, the method raises an exception.

- **_flutterPubGet**: Automatically installs dependencies listed in the *pubspec.yaml* file by running `flutter pub get`.

- **_addTestDependency**: Adds the Flutter *test* package as a dependency using `flutter pub add test`.

- **create_test**: Creates a test file in the designated *test* directory of the project. If the file already exists and overwriting is not allowed, an exception is raised.

- **get_test_content**: Retrieves the content of a test file from the *test* directory.

- **run_test**: Executes a specified Dart test file using the Flutter CLI and returns the results.

- **validate**: Iterates through all Dart test files in the *test* directory and validates them by running each test.

- **getListSourceFiles**: Collects and returns a list of all source files in the *lib* directory, ensuring that the *main.dart* file is prioritized as the entry point.

This design enables seamless integration of Flutter-specific features into the **Test Genie** system while adhering to the modular structure defined by the **Project** class. By implementing these methods, the **Flutter** class ensures compatibility with the broader system and provides developers with a streamlined process for managing and testing Flutter projects.

4.2 Business Logic Analyzer module

The **Business Logic Analyzer** module is designed to parse and analyze the source code of a project. It constructs a **Dependency Diagram** that represents the logical structure and relationships within the project. By leveraging framework-specific analysis strategies, such as the *FlutterAnalyzeStrategy*, this module identifies functional blocks and their interconnections. Each block is further enriched with predictions generated by the **AI Agent**, which analyzes the code to provide insights into its behavior and logic. This modular design allows the Business Logic Analyzer to be easily extended to support additional frameworks, making it versatile and scalable for various software projects. The output of this module serves as the foundation for the subsequent test generation process.

4.2.1 DependencyDiagram class

The **DependencyDiagram** class serves as a connector within the Test Genie system, bridging the gap between the framework-specific analysis strategies, such as *FlutterAnalyzeStrategy*, and the AI-powered prediction functionality provided by the **AI_Agent**. This class is responsible for constructing a logical representation of the project in the form of a dependency diagram, which comprises blocks (representing functional units) and connections (representing the relationships between those units). The **_generateDiagram** method encapsulates this functionality by invoking the appropriate analysis strategy for the project's framework (Listing A.3), allowing the class to dynamically adapt to diverse frameworks supported by the system. This modular design ensures that the class is both extensible and maintainable as new frameworks are introduced.

In addition to structural analysis, the **DependencyDiagram** class leverages the **AI_Agent** to enrich the diagram with meaningful predictions. Through the **_getPredictions** method (Listing A.3), each block in the diagram is analyzed to generate insights into its behavior and logic, which are subsequently embedded into the block. This integration of AI-based predictions and static code analysis makes the **DependencyDiagram** a powerful tool for understanding the project's overall architecture and behavior. By combining these two mechanisms, this class plays a pivotal role in preparing the business logic analyzation for further steps in the Test Genie system: test generation and validation.

Diagram objects

The DependencyDiagram class relies on objects from the Diagram folder to represent the blocks and connections within the dependency structure. These objects are defined as follows:

- **Block class**: Represents the functional units of the source code, such as files, classes, functions, or variables (Listing A.4). Each block contains the following attributes:
    - *name*: The name of the block.
    - *content*: The source code or content of the block.
    - *type*: The type of the block, determined by the BlockType class.
    - *prediction*: (Optional) AI-generated predictions for the block's logic or behavior.

  Additionally, the Block class provides methods such as:
    - **getContentNoComment**: Removes comments from the block's content for clean analysis.
    - **setPrediction** and **getPrediction**: Manage predictions for the block.
- **BlockType class**: Enumerates the possible types of blocks (Listing A.5), such as *File*, *Class*, *Function*, and more. It also provides methods to:
    - Generate database queries for storing and managing block types.
    - Define the schema for the BlockType database table.
- **Connection class**: Represents relationships between blocks (Listing A.6), with attributes:
    - *head*: The source block of the connection.
    - *tail*: The destination block of the connection.
    - *type*: The type of relationship, determined by the ConnectionType class.

  It also facilitates database storage and retrieval through schema definitions.
- **ConnectionType class**: Enumerates the types of relationships between blocks (Listing A.7), such as *Extend*, *Implement*, *Call*, and *Import*. It provides similar database-related methods as the BlockType class.

FlutterAnalyzeStrategy Algorithm

The **FlutterAnalyzeStrategy** function (Listing A.8) is a core component of the **DependencyDiagram** generation process within the **Business Logic Analyzer** module. This function is specifically designed to analyze Flutter projects by reading their source code, breaking it into logical units (*blocks*), and appending these blocks to the diagram. It employs three custom backtracking algorithms (*ImportAnalyzer*, *ContainAnalyzer*, and *CallAnalyzer*) to achieve a comprehensive structural and relational analysis of the project. Here's a detailed breakdown of the algorithm:

- **Initialization:**
  - The function begins by retrieving the list of source files in the project using the **getListSourceFiles** method from the **Project** class (Listing A.1).
  - The first file in the list is assumed to be the project's entry point (typically `main.dart`). Its content is extracted, and a new **Block** object is created to represent it. This block is assigned the `FILE` type from the **BlockType** class.
  - The newly created `main.dart` block is appended to the **blocks** attribute of the **DependencyDiagram** instance.
- **Import Analysis (ImportAnalyzer):**
  - The **ImportAnalyzer** algorithm (Listing A.9) scans the content of the `main.dart` block for `import` statements. These statements indicate dependencies on other Dart libraries or files.
  - For each `import` statement, a **Connection** object is created between the current block (as the *head*) and the imported file (as the *tail*). The connection type is marked as `IMPORT`.
  - Unlike the other analyzers, **ImportAnalyzer** primarily focuses on establishing file-level relationships and does not create new blocks.
- **Containment Analysis (ContainAnalyzer):**
  - The **ContainAnalyzer** algorithm (Listing A.10) dives deeper into each file to identify hierarchical relationships within the code. For example:
    * Classes contained within files.
    * Standalone functions contained within files.
    * Functions and attributes contained within classes.
  - For each identified entity, a new **Block** object is created and appended to the **blocks** list. The type of the block is determined based on the entity, such as `CLASS`, `FUNCTION`, or `CLASS_ATTRIBUTE`.
  - Connections are established between the parent block (e.g., the file block) and the contained entities, using the `CONTAIN` relationship type.
- **Call Analysis (CallAnalyzer):**
  - The **CallAnalyzer** algorithm (Listing A.11) identifies calling activities between functions and classes. For instance:
    * Functions calling other functions, either within the same file or across files.
    * Methods from one class invoking methods or attributes of another class.
  - For each calling activity found, a **Connection** object is created to represent the caller (as the *head*) and the callee (as the *tail*). The relationship type for these connections is set to `CALL`.
  - This analysis also considers cross-file and cross-class interactions, providing insights into the dynamic flow of the project.

- **Finalizing the Diagram:**
  - After executing the three algorithms, the **blocks** list of the **DependencyDiagram** instance contains a comprehensive representation of the project's structural elements.
  - Similarly, the **connections** list captures the relationships between these elements, making the diagram a complete and versatile model of the project's dependencies and interactions.

The **FlutterAnalyzeStrategy** function effectively combines the results of these three backtracking algorithms to deliver a detailed and accurate dependency diagram. By modularizing the analysis into distinct phases (*Import Analysis*, *Containment Analysis*, and *Call Analysis*), the function ensures that the structural and relational aspects of the project are thoroughly captured. This makes it an indispensable part of the Test Genie system's ability to analyze and generate tests for Flutter projects.

4.2.2 AI_Agent class

The **AI_Agent** class is a component of the Test Genie system that provide AI-driven insights into the business logic of analyzed code blocks using *Langchain framework* [5]. This class is initialized within the **DependencyDiagram** class and utilized in the *_getPredictions* method (Listing A.3) to generate structured predictions for each block. The initialization process of the **AI_Agent** involves setting up its environment, loading necessary resources, and preparing the underlying AI models and vector stores.

Initialization Flow

The initialization of the **AI_Agent** class (Listing A.12) involves several key steps to prepare its environment and components:

- **Environment Setup:**
  - The class begins by loading environment variables from a `.env` file using the `load_dotenv` function. If the file fails to load, an exception is raised.
  - Critical environment variables (Listing A.13) include:
    * `BASE_URL`: The base URL for API requests.
    * `BLA_LLM_MODEL`: The name of the language model used for predictions.
    * `EMBED_MODEL`: The embedding model used for vectorization.
- **Model and Embedding Initialization:**
  - A **ChatOpenAI** instance is initialized for interacting with the language model. This instance is configured with the `BASE_URL` and `BLA_LLM_MODEL`.
  - An **OpenAIEmbeddings** instance is initialized for generating document embeddings. It is configured to skip context length checks for compatibility with specific setups.
- **Vector Store Creation:**

- The **AI_Agent** manages document vector stores for efficient retrieval. A predefined list of documents (e.g., `flutter_tutorial.pdf`) is used to populate these stores.
- For each document:
  * The document is loaded using appropriate loaders (e.g., **PyPDFLoader**).
  * The document is split into chunks using the **SentenceTransformersTokenTextSplitter**.
  * A persistent vector store is created for the document using the **Chroma** library.
- If a vector store already exists for a document, it is reused without reinitialization.

- **Retriever Initialization:**
  - For each vector store, a **retriever** is configured to fetch relevant documents based on similarity thresholds. These retrievers are stored for later use.

- **Agent Initialization:**
  - The *_agent_init* method is invoked to set up a history-aware retrieval system and define the agent's behavior for analyzing code.
  - Custom prompts are created for contextualizing queries and for generating predictions. These prompts guide the language model in providing detailed business logic analysis and testing scenarios.
  - A **react_agent** is created using the **create_react_agent** function, and its execution is managed by an **AgentExecutor**.

generate_BLA_prediction Function

The **generate_BLA_prediction** function (Listing A.12) is the primary method of the **AI_Agent** class, responsible for analyzing source code and generating structured predictions. It takes two input parameters: `source_code`, which is the code snippet to be analyzed, and `chat_history`, a list of previous interactions that provide context for the analysis. These inputs allow the function to understand the context of the code and any prior discussions related to it.

The function begins by invoking the **agent_executor**, which analyzes the source code in the context of the provided chat history. The executor uses the retrievers and the language model to perform an initial analysis, retrieving any relevant documents from the vector stores to assist in understanding the code. This step produces a preliminary output that captures the essential details of the code's functionality and properties.

Once the initial analysis is complete, the function refines the output by directly querying the language model with a structured prompt. This prompt is specifically designed to guide the model in organizing the analysis into well-defined sections. These sections include a brief explanation of what the code does, an assessment of its testability, and a set of detailed testing scenarios. The testing scenarios are formatted to clearly describe the functionality being tested, the input values, and the expected outcomes. This ensures that the generated test cases are practical, comprehensive, and easy to understand.

The final structured response produced by the language model includes three main components. The first is a **Brief Explanation**, summarizing the purpose and functionality of the code. The second is a **Testability Assessment**, evaluating whether the code can be tested and identifying the appropriate types of tests, such as unit, widget, or integration tests. The third component is a list of **Testing Scenarios**, which outlines specific test cases with descriptive names, input values, and expected behaviors. These scenarios ensure coverage of normal cases, edge cases, and special conditions, providing a thorough basis for test planning.

The structured response generated by the *generate_BLA_prediction* function serves as the final output, ready to be integrated into the **DependencyDiagram** for further use. By combining retrieval-augmented generation with precise prompts, this function delivers high-quality insights that enable efficient and accurate test planning for Flutter/Dart projects.

4.2.3 Test Generator Module

The **Test Generator** module is a core component of the Test Genie system, responsible for generating test cases based on predictions and code details. The **Test_Generator** class orchestrates this process and ensures seamless test case generation using the LangChain framework. Below, we describe the flow of the **Test_Generator** class and its primary operations, along with the complete test generation and validation process.

Initialization Flow

The initialization of the **Test_Generator** class involves several critical steps to set up its environment and components (Listing A.13):

- **Environment Setup:** The class begins by loading environment variables from a `.env` file using the `load_dotenv` function. If the file fails to load, an exception is raised. Key variables include:
  - `BASE_URL`: The base URL for API requests.
  - `TG_LLM_MODEL`: The name of the language model used for generating test cases.
  - `EMBED_MODEL`: The embedding model used for document vectorization.
- **Model and Embedding Initialization:** A **ChatOpenAI** instance is created to interact with the language model, configured with the `BASE_URL` and `TG_LLM_MODEL`. Additionally, an **OpenAIEmbeddings** instance is initialized for generating document embeddings, with specific configurations to handle compatibility issues.
- **Vector Store Creation:** The class manages document vector stores to facilitate efficient retrieval:
  - A predefined list of documents is retrieved using the `_getStoreList` function.
  - For each document, the content is loaded using appropriate loaders (e.g., **PyPDFLoader**), split into chunks using the **SentenceTransformersTokenTextSplitter**, and stored in a persistent vector store using the **Chroma** library.

  – Existing vector stores are reused to avoid redundant computations.

- **Retriever Initialization:** Each vector store is wrapped in a retriever, which is configured to fetch relevant documents based on similarity thresholds. These retrievers are stored for use during test case generation.

- **Error Handling Setup:** To handle errors during test generation:

  – An error cache (`error_fix_cache`) is created to store previously fixed errors.
  – A set of attempted fixes (`attempted_fixes_for_error`) is maintained to track retry attempts.
  – A maximum retry limit (`max_error_fix_attempts`) is defined to prevent infinite loops.

Test Generation and Validation Flow

The **generateTest** function in the `main.py` file (Listing A.15) orchestrates the full test generation and validation process. This process involves generating a test case, validating it, and iteratively fixing errors until the test passes or a retry limit is reached.

The process begins with a POST request containing the following parameters:

- `git_url`: The Git repository URL of the project under analysis.
- `block_id`: The identifier of the code block for which a test case is being generated.

The backend first uses the **getDBMS** function to initialize a **DBMS** instance. This instance manages the project's metadata and provides access to block-level information. The **Test_Generator** class is then instantiated to handle the test generation process.

The test generation begins with a call to the **generate_test_case** method of the **Test_Generator** class. This method accepts the following inputs:

- `package_name`: The name of the project's package, used for import statements.
- `code_location`: The file path of the block's original code.
- `function_name_and_arguments`: The function signature, including its name and arguments.
- `prediction`: A description of the block's behavior, typically generated by the Business Logic Analyzer.

The method generates a structured test case that includes necessary import statements and assertions to validate the function's behavior under various conditions.

Once the test case is generated, it is saved to a file (e.g., `block_x_test.dart`) using the **Project** class's `create_test` method. The backend then initiates the validation process by running the test file using the `run_test` method of the **Project** class.

If errors occur during the test execution, the backend invokes the **fix_generated_code** method of the **Test_Generator** class. This method uses the error message, function details, and prediction to iteratively refine the test case. The updated test case is saved and re-executed. This process repeats until the test passes or the retry limit (default: 5 iterations) is reached.

Upon successful validation, the generated test case content is returned in the response, along with a success message. If the validation fails after exhausting the retry limit, an error message is returned.

Integration with the Test Genie System

The **generateTest** function demonstrates the seamless integration of the **Test_Generator** class with the Test Genie system. By combining predictions from the **Business Logic Analyzer** with iterative test case refinement, the process ensures that the generated tests are both functional and robust. This end-to-end flow automates the testing process, reducing manual effort and improving accuracy.

4.3 Other implementations

Beside core modules, the Test Genie system also includes other component to help manage data and help users interact with the system. Like every other web application, Test Genie also has a API request handler, a database management module (DBMS) and a frontend interface.

- **API request handler**: This component is responsible for handling all the API requests from the frontend. It will receive the request, invoke needed calculation method and return the response to the frontend. The API request handler is built using Flask framework due to its lightweight and easy to use. The API request handler will also handle authentication and authorization for the system.
- **DBMS**: To save blocks and block predictions generated from BLA module, Test Genie need a database to store the data. DBMS module is responsible for this task, connecting services with MySQL database.
- **Frontend**: This component allow users to interact with the system, sending requests to backend and visualize the connections between blocks. The frontend is built using React framework, which is a popular choice for quickly and efficiently building user interfaces.

4.3.1 DBMS module

The **DBMS module** is component responsible for managing the database and facilitating interactions between the project's metadata and the database. It provides functionalities for storing, retrieving, and updating information related to blocks, connections, and projects in the system. The **DBMS** module is composed of the **DBMS** class and the **Table** class, each serving specific purposes in database management.

DBMS Class

The **DBMS** class (Listing A.21) is the primary interface for interacting with the database. It handles database initialization, project insertion, and retrieval of blocks and connections. This class is initialized with a **Project** instance and ensures that the database is ready for operations. The initialization flow involves checking whether the database is initialized using the `_isDBinit` method. If not, the `_initDB` method is called to create necessary tables and populate them with default values.

When a new project is added, the **DBMS** class verifies if the project already exists in the database using the `_isProjectExistInDB` method. If the project does not exist, the `_insertProject` method maps the project's blocks and connections into the database. Blocks and connections are extracted from a **DependencyDiagram** instance associated with the project. These are then stored in respective tables using the `_mapBlocksIntoDB` and `_mapConnectionsIntoDB` methods.

The class also provides methods for retrieving and updating data:

- `getBlockName`, `getBlockContent`, and `getBlockPrediction` retrieve information about blocks based on their IDs.

- `getBlockOriginalFile` performs backtracking to identify the original file associated with a block.

- `updateBlockPrediction` updates a block's prediction in the database.

All database queries are executed through the `execute` method, which establishes a connection, executes the query, and closes the connection. The **DBMS** class employs a modular design, relying on the **Table** class for creating and executing SQL queries dynamically.

Table Class

The **Table** class (Listing A.16) provides a flexible and reusable framework for managing database tables. It abstracts SQL query generation, enabling the **DBMS** class to focus on higher-level operations. Each instance of the **Table** class represents a database table, defined by its name and columns.

The **Table** class supports the following operations:

- **Table Creation:** The `getCreateSQL` method generates an SQL query to create the table if it does not already exist. Column definitions are specified during initialization.

- **Data Retrieval:** The `getSelectSQL` method constructs SQL queries for retrieving data. It supports both conditional and unconditional retrieval.

- **Data Insertion:** The `getInsertSQL` method generates SQL queries for inserting data into the table. Column names and values are supplied as dictionaries.

- **Data Update:** The `getUpdateSQL` method constructs SQL queries for updating existing records. Conditions and values are specified as dictionaries.

The **Table** class is heavily utilized by the **DBMS** class for creating and managing tables such as `Block` (Listing A.18), `Connection` (Listing A.20), `BlockType` (Listing A.17), and `ConnectionType` (Listing A.19). By encapsulating SQL query generation, the **Table** class ensures consistency and reduces redundancy across the system.

Integration and Workflow

Together, the **DBMS** and **Table** classes form a cohesive system for managing the Test Genie database. The **DBMS** class utilizes the **Table** class to dynamically generate SQL queries, enabling seamless interaction with the database. This architecture ensures that the database remains synchronized with the project's metadata, providing a reliable foundation for the Test Genie system's operations.

4.3.2 Backend - API implementation

The **Backend API implementation** serves as the interface between the frontend and the core functionalities of the Test Genie system. It is implemented using the Flask framework, allowing for seamless communication via HTTP requests. The backend exposes multiple API endpoints for managing projects, generating dependency diagrams, retrieving block details, updating predictions, and generating test cases. Below is a detailed analysis of the key API endpoints provided in the `main.py` file (Listing A.15).

API Endpoints

- **/createProject (POST):** This endpoint is responsible for initializing a new project in the system. The request must include a JSON payload with the `git_url` of the project repository. Upon receiving the request, the backend clones the repository and creates a **Project** instance. The response confirms the successful creation of the project.

- **/getDiagram (POST):** This endpoint generates a dependency diagram for the specified project. The request must include the `git_url` of the project. The backend creates a **DBMS** instance for the project and retrieves the diagram in JSON format using the `getJsonDiagram` method. The response includes the diagram's blocks and connections, which represent the project's logical structure.

- **/getBlockContent (POST):** This endpoint retrieves the content of a specific block in the project. The request must include `git_url` and `block_id`. The backend uses the `getBlockContent` method of the **DBMS** class to fetch the block's content from the database and returns it as a response.

- **/getBlockPrediction (POST):** This endpoint retrieves the AI-generated prediction for a specific block. Similar to `/getBlockContent`, the request must include `git_url` and `block_id`. The backend uses the `getBlockPrediction` method to fetch the prediction from the database and returns it.

- **/getBlockDetail (POST):** This endpoint provides comprehensive details about a block, including its content, prediction, and associated test file content (if available). The request must include `git_url` and `block_id`. The backend combines the results of `getBlockContent`, `getBlockPrediction`, and test file retrieval methods to provide a detailed response.

- **/updateBlockPrediction (POST):** This endpoint updates the prediction of a specific block in the database. The request must include `git_url`, `block_id`, and the new `prediction`. The backend uses the `updateBlockPrediction` method of the **DBMS** class to update the database. A success message is returned upon completion.

- **/generateTest (POST):** This endpoint orchestrates the process of generating a test case for a specific block. The request must include `git_url` and `block_id`. The backend performs the following steps:

  1. Initializes a **DBMS** instance and a **Test_Generator** instance.
  2. Calls the `generate_test_case` method of the **Test_Generator** class to create a test case based on the block's prediction and metadata.

3. Saves the generated test case to a file using the **Project** class's `create_test` method.
4. Validates the test case by running it via the `run_test` method.
5. If errors are encountered, iteratively refines the test case using the `fix_generated_code` method until the test passes or a retry limit is reached.

The response includes the generated test file content and a success message upon successful execution.

Error Handling

The backend API includes robust error handling mechanisms to ensure stability and reliability:

- Missing or invalid request parameters result in a clear error message being returned to the client.
- Methods like `run_test` and `create_test` include exception handling to manage file creation and test execution errors.
- The `/generateTest` endpoint employs a retry mechanism to iteratively refine failing test cases, ensuring a high success rate.

Integration with Core Modules

The API endpoints rely on the core modules of the Test Genie system:

- The **Project Manager** module is used for project initialization and file management.
- The **Business Logic Analyzer** module provides block predictions that guide test generation.
- The **Test Generator** module handles test case creation and validation.
- The **DBMS** module manages the database, ensuring consistent storage and retrieval of project metadata.

By leveraging these modules, the backend API provides a comprehensive interface for users to interact with the Test Genie system, facilitating efficient project management, dependency analysis, and test generation

4.3.3 Frontend implementation

The **Frontend** of the Test Genie system is responsible for providing a user-friendly interface to interact with the backend services. It is built using a React framework, leveraging components, routing, and services to achieve a modular and maintainable architecture. Below is an analysis of its structure and loading logic.

Directory Structure

The frontend source code is organized into the following key files and directories:

- **`App.js` and `App.css`:** Serves as the entry point for the React application, rendering the main application structure and applying global styles.

- **`index.js` and `index.css`:** Initializes the React application and injects it into the DOM. The `index.css` file applies global styles.

- **`pages/`:** Contains React components representing individual pages in the application. Each page corresponds to a specific route.

- **`routes/`:** Manages routing logic for the application, connecting URLs to their respective page components.

- **`services/`:** Includes utility functions and services for making API calls to the backend. This modularizes and centralizes the API interaction logic.

- **`setupTests.js`:** Configures the testing environment for the frontend, ensuring proper setup for unit and integration tests.

- **`reportWebVitals.js`:** Used for measuring the application's performance metrics.

- **`logo.svg`:** Contains assets like the application logo used across the frontend interface.

- **`App.test.js`:** Includes test cases for the main application component.

Loading Logic

The frontend follows a typical React application lifecycle with the following loading logic:

1. When the application starts, `index.js` initializes the React application and renders the root `App.js` component into the DOM.

2. The `App.js` component manages the high-level structure of the application, including the navigation bar, footer, and main content area.

3. The `routes/` directory defines the mapping between URLs and React components, ensuring that the correct page is displayed for each route.

4. Components in the `pages/` directory dynamically render content based on the application's state and user interactions.

5. The `services/` directory provides reusable functions for making API calls to the backend. These functions are used within page components to fetch or send data.

6. Global styles and assets are applied using `App.css` and `index.css`, ensuring a consistent look and feel throughout the application.

Integration with Backend

The frontend interacts with the backend API endpoints via functions provided in the `services/` directory. These functions encapsulate HTTP requests, allowing components to focus on rendering logic without worrying about API interaction details. This separation of concerns improves maintainability and reduces code duplication.

Testing and Performance

The frontend includes a testing configuration file, `setupTests.js`, and test cases, such as `App.test.js`, to ensure the reliability of components. Performance metrics can be measured using the `reportWebVitals.js` file, providing insights into the application's runtime behavior.

In summary, the frontend implementation of the Test Genie system is modular, maintainable, and well-integrated with the backend services, offering a seamless user experience for interacting with the system.

4.4 Implementation Result - Demo

This section provides a detailed overview of the Test Genie system's user interface and functionality as demonstrated in the application. Screenshots from the application are included to showcase its features, explain the purpose of each button, and describe how users can interact with the system.

4.4.1 Homepage

The homepage (Figure 4.1) serves as the entry point for users interacting with the system. It provides a user-friendly interface where users can initiate various actions such as uploading a project or exploring previously analyzed projects.

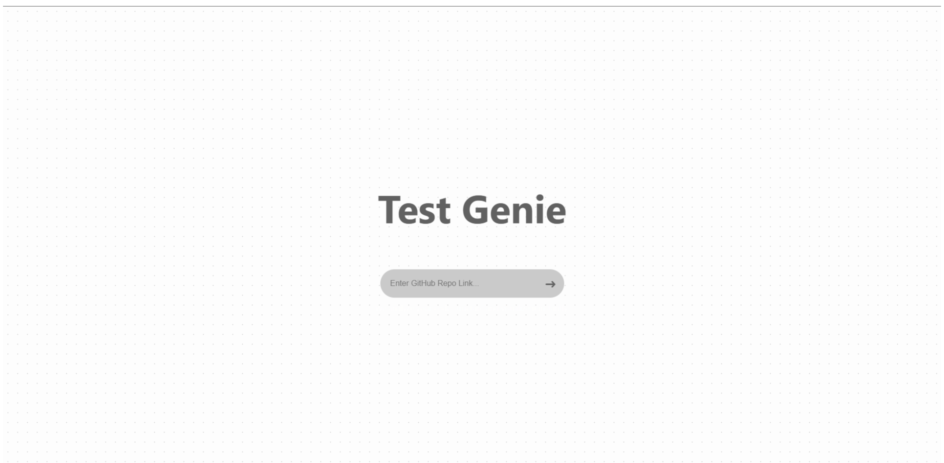

Figure 4.1: Homepage of Test Genie system.

Main Features:

- **Git project repo link input field:** Users can input the URL of their Git project repository to analyze.

4.4.2 Interactive Dependency Diagram

The dependency diagram is one of the core components of the Test Genie system. It visualizes to users the relationships between blocks in a project, such as files, functions, and classes.

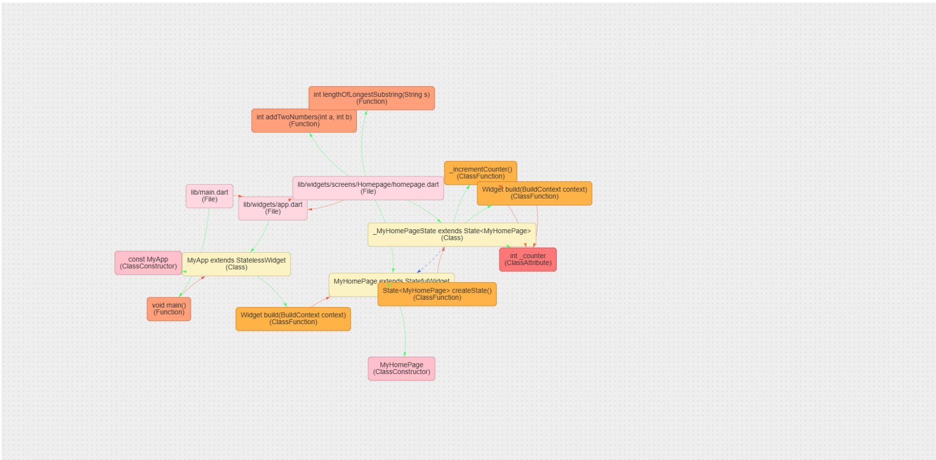

Figure 4.2: Initial load of the dependency diagram.

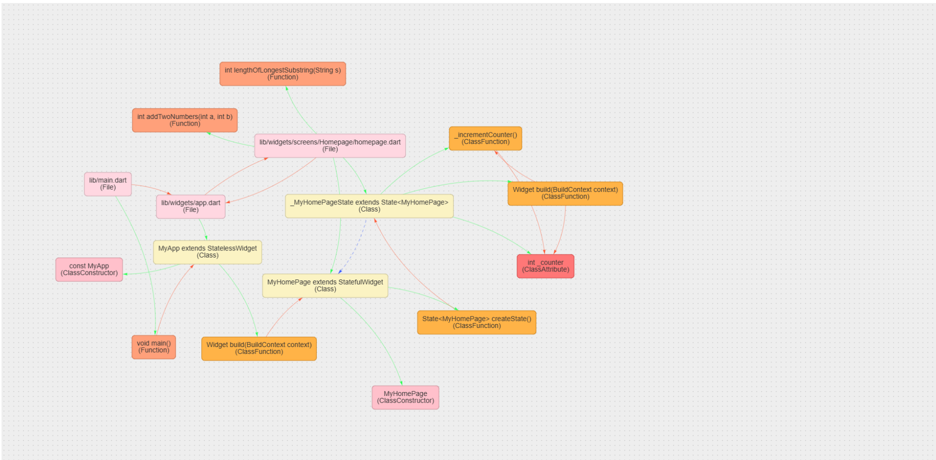

Figure 4.3: Diagram blocks can be dragged to rearrange their positions.

Key Interactions:

- **Drag and Drop:** Users can click and drag blocks to rearrange them for better visualization (Figure 4.3).
- **Zoom and Pan:** The diagram supports zooming in and out for detailed or high-level views. Users can pan the diagram to focus on specific areas.
- **Click on Block:** Clicking a block opens its details, including content (source code), predictions, and test cases.

4.4.3 Block Detail View

When a block in the diagram is clicked, the Block Detail View is displayed (Figure 4.4). This view provides detailed information about the selected block.

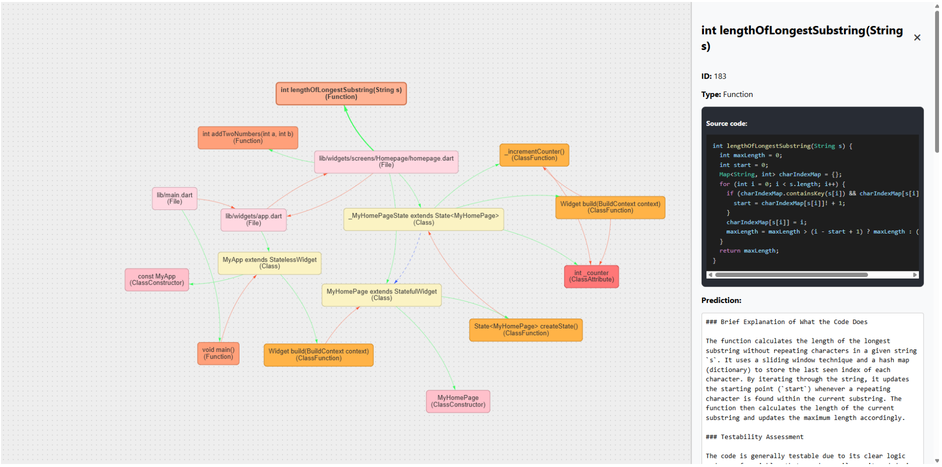

Figure 4.4: Block Detail View showcasing the block's content and prediction.

Key Features:

- **Source Code Content:** Displays the source code content of the block.
- **Prediction:** Shows the AI-generated prediction for the block's functionality.
- **Adjustable Prediction:** Users can modify the prediction to refine its accuracy.
- **Test Case Viewer:** Displays the associated test cases for the block, if available.

4.4.4 Adjustable Predictions

One of the unique features of Test Genie is the ability to adjust AI-generated predictions. Figure 4.5 showcases how users can interact with and modify predictions.

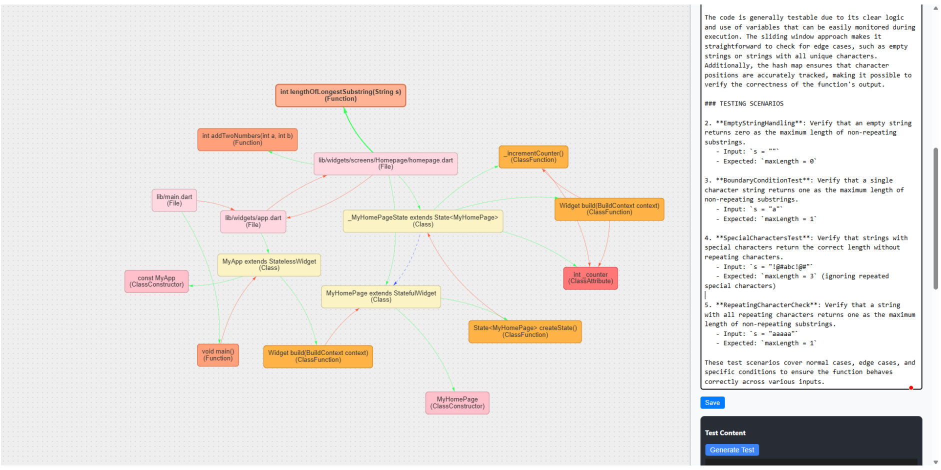

Figure 4.5: Prediction adjustment interface for refining AI-generated predictions.

Key Features:

- **Editable Text Field:** Users can directly edit the prediction to increase its accuracy.
- **Save Button:** Saves the updated prediction to the database.

4.4.5 Test Generation

After analyzing the blocks, Test Genie allows users to generate test cases for specific blocks. Users can view the generated test cases and validate them.

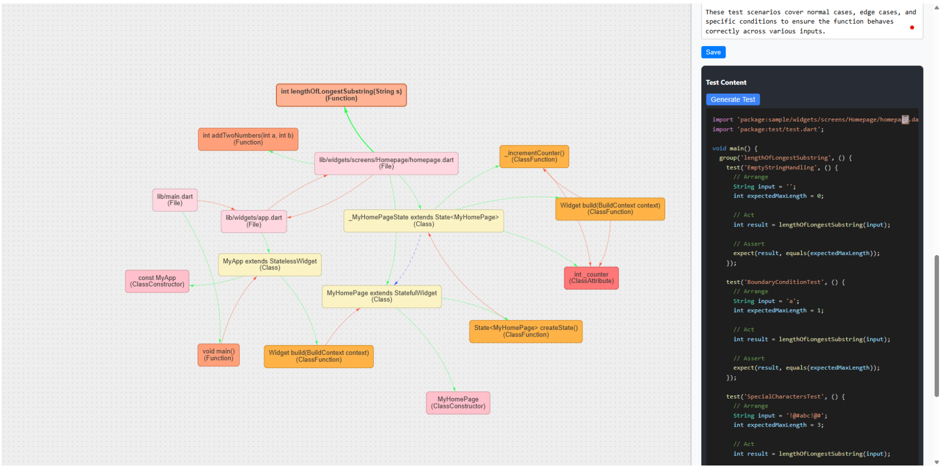

Figure 4.6: Generated test cases for a specific block.

Key Features:

- **View Test Cases:** Displays the generated test cases for the selected block.
- **Quick copy button:** Allows users to quickly copy the test case code to the clipboard for easy integration into their projects.

4.4.6 Summary of Interactions

The Test Genie system combines a user-friendly interface with advanced functionalities to simplify the process of dependency analysis and test generation. By integrating interactive diagrams, adjustable predictions, and robust test generation, it provides a comprehensive solution for software analysis and testing.

Chapter 5
DISCUSSION AND EVALUATION

This chapter presents a comprehensive evaluation of the **Test Genie** system, analyzing its performance characteristics, accuracy in generating test files, and comparing it with existing approaches. The evaluation aims to assess whether the system meets the requirements established in Chapter 3 and to identify both strengths and limitations of the implemented solution. By examining metrics related to execution time, algorithmic complexity, and test generation accuracy, this chapter provides insights into the practical viability of using AI-driven techniques for automated test generation in Flutter projects.

5.1 Performance Analysis

The performance of the **Test Genie** system is evaluated based on two most complex process: AI generation time and BLA algorithm complexity. Others processes are negletable since their time complexity are O(1). Although DBMS module is dependent on the server's response time, it is impossible to estimate the time complexity of the server's response. However, it is important to note that the DBMS module may become a bottleneck in the overall system performance.

To fully calculate the performance estimation of this system, let the estimation time for AI to generate tests is T_{AI_test} and the time to fully generate blocks in BLA is T_{BLA}. Since the block generating procedure contain two separate steps are source code splitting and blocks's prediction generation, we can define the time complexity of BLA as $T_{BLA} = T_{split} + T_{predict}$, where T_{split} is the time complexity of source code splitting and $T_{predict}$ is the time complexity of blocks's prediction generation. The overall time complexity of the system can be expressed as:

$$T_{total} = T_{AI\_test} + T_{BLA} = T_{AI\_test} + T_{split} + T_{predict} \tag{5.1}$$

If we account for DBMS module as a parameter m, the overall time complexity of the system can be expressed as:

$$T_{total} = T_{AI\_test} + T_{BLA} + m = T_{AI\_test} + T_{split} + T_{predict} + m \tag{5.2}$$

5.1.1 Code Splitting Algorithm complexity

5.1.2 AI generation time estimation

5.2 Accuracy Evaluation

5.3 Comparison with Other Approach

Chapter 6
CONCLUSION AND FUTURE WORK

6.1 Conclusion

6.2 Future Work

References

[1] Capgemini - "World Quality Report 2021-22", Thirdteenth edition.

[2] Glassdoor - "Qa Tester Salaries", https://www.glassdoor.com/Salaries/qa-tester-salary-SRCH_KO0,9.htm

[3] Herb Krasner, Consortium for information & Software quality - "The Cost of Poor Software Quality in the US: A 2020 Report".

[4] Migual Gringerg - "Flask Web Development", 2014, Published by O'Reilly Media, Inc.

[5] "Langchain", https://python.langchain.com/docs/introduction/

[6] Abhishek Singh - "Essential Python for Machine Learning", new edition 2023.

[7] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, Pankaj Jalote - "Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools"

[8] Tosin Adewumi, Nudrat Habib, Lama Alkhaled & Elisa Barney ML Group, EISLAB, Luleå University of Technology, Sweden - "On the Limitations of Large Language Models (LLMs): False Attribution".

[9] Harry M. Sneed, Katalin Erdos - "Extracting Business Rules from Source Code", April 1996

[10] Adrian S. Barb, Colin J. Neill, Raghvinder S. Sangwan, Michael J. Piovoso - "A statistical study of the relevance of lines of code measures in software projects", May 7, 2014.

[11] Bertolino, Antonia - "A Historical Perspective on Software Testing", IEEE Annals of the History of Computing, vol. 31, no. 3, pp. 32-43, 2018.

[12] McMinn, Phil - "A Survey of Search-Based Software Testing", Software Testing, Verification and Reliability, vol. 24, no. 4, pp. 297-321, 2014.

[13] Watson, James; Singh, Aditya; Patel, Priya - "BenchmarkLLM: A Comparative Analysis of LLM Performance in Software Testing Tasks", Proceedings of the International Conference on Software Engineering, pp. 1231-1242, 2023.

[14] Johnson, Sarah; Ahmed, Khalid; Brown, Thomas - "Comparing LLM-based and Traditional Test Case Generation: Metrics, Performance, and Practical Applicability", ACM Transactions on Software Engineering and Methodology, vol. 32, no. 2, pp. 1-27, 2023.

[15] Wang, Xiaofeng; Liu, Yuanchao; Zhang, Min - "Understanding the Limitations of Large Language Models in Software Testing", IEEE Transactions on Software Engineering, vol. 49, no. 6, pp. 1243-1262, 2023.

[16] Li, Wei; Zhang, Jia - "Challenges in Testing Flutter Applications: An Empirical Study", Proceedings of the International Conference on Mobile Software Engineering and Systems, pp. 112-123, 2022.

[17] Patel, Ravi; Gupta, Anjali; Mehta, Sanjay - "Enhancing LLM-based Test Generation through Framework-Specific Fine-tuning", Journal of Systems and Software, vol. 188, pp. 111-128, 2023.

[18] von Halle, Barbara - "Business Rules Applied: Building Better Systems Using the Business Rules Approach", Wiley & Sons, 2001.

[19] Morgan, Tony - "A Framework for Business Rule-driven Software Development", Journal of Information Technology Management, vol. 13, no. 3, pp. 22-41, 2002.

[20] Premerlani, William; Blaha, Michael - "Manual Extraction of Business Rules from Legacy Code: Process and Challenges", IEEE Software, vol. 11, no. 4, pp. 77-87, 1994.

[21] Fisher, Martin; Jin, Daqing; Rothermel, Gregg - "Identifying and Extracting Business Rules from Object-Oriented Systems", Proceedings of the Conference on Software Maintenance and Reengineering, pp. 240-249, 2002.

[22] Zou, Ying; Lau, Terence - "Dynamic Analysis for Business Rule Extraction in Event-Driven Systems", Information Systems Journal, vol. 17, no. 3, pp. 321-346, 2007.

[23] Wong, William; Menzies, Tim - "Machine Learning Approaches for Business Logic Extraction", Proceedings of the International Conference on Automated Software Engineering, pp. 144-155, 2019.

[24] Chen, Xiaohong; Wang, Lei; Zhou, Yang - "Hybrid Business Logic Extraction: Combining Static Analysis, Dynamic Tracing, and Machine Learning", Journal of Software: Evolution and Process, vol. 32, no. 5, pp. e2231, 2020.

[25] Rodriguez, Carlos; Baez, Marcos; Daniel, Florian - "Business Logic Consistency Challenges in Microservice Architectures", Proceedings of the International Conference on Service-Oriented Computing, pp. 301-317, 2021.

[26] Garcia, David; Cabot, Jordi - "Business Rule Extraction in Event-Driven Architecture: Challenges and Solutions", Software & Systems Modeling, vol. 20, no. 3, pp. 733-754, 2021.

[27] Nguyen, Tuan; Kim, Moonzoo - "Analysis of Business Logic Distribution in Mobile UI Frameworks with Focus on Flutter", Proceedings of the International Conference on Mobile Software Engineering and Systems, pp. 54-65, 2022.

[28] Inozemtseva, Laura; Holmes, Reid - "Coverage Is Not Strongly Correlated with Test Suite Effectiveness", Proceedings of the International Conference on Software Engineering, pp. 435-445, 2014.

[29] Papadakis, Mike et al. - "A Systematic Literature Review of How Mutation Testing Supports Test Activities", Empirical Software Engineering, vol. 24, no. 2, pp. 1665-1706, 2019.

[30] Athanasiou, Dimitrios; Nugroho, Ariadi; Visser, Joost - "Measuring and Improving Test Suite Maintainability: A Case Study", Proceedings of the International Conference on Software Testing, Verification and Validation, pp. 88-99, 2014.

[31] Bavota, Gabriele et al. - "On the Readability of Test Code: A Study on the Relationship between Readability and Test Suite Quality", Journal of Systems and Software, vol. 117, pp. 48-66, 2016.

[32] Tufano, Michele et al. - "A Comparative Study of Human-Written and AI-Generated Test Cases", Proceedings of the International Conference on Automated Software Engineering, pp. 132-143, 2022.

[33] Martin, Andrea; Patel, Sunil - "A Comprehensive Guide to Flutter Testing: Strategies and Best Practices", Mobile Software Engineering and Development, vol. 8, no. 2, pp. 89-112, 2022.

[34] Shah, Rajan; Patel, Vimal; Desai, Anand - "Common Challenges and Solutions in Flutter Unit Testing: An Empirical Study", Proceedings of the Asia-Pacific Software Engineering Conference, pp. 201-210, 2022.

[35] Zhao, Lina; Li, Jianming - "Understanding the Challenges of Widget Testing in Flutter Applications", Journal of Software: Evolution and Process, vol. 34, no. 5, pp. e2412, 2022.

[36] Martinez, Fernando; Leiva, Luis - "An Analysis of Platform-Specific Issues in Flutter Applications", Proceedings of the International Conference on Mobile Software Engineering and Systems, pp. 78-89, 2023.

[37] Kim, Sungjae; Park, Jaehyun; Lee, Sukyoung - "Platform-Aware Test Generation for Cross-Platform Mobile Applications", IEEE Transactions on Software Engineering, vol. 48, no. 8, pp. 2890-2907, 2022.

Appendix A
LISTINGS

```
import os
import subprocess

class Project:

    _framework = ''
    def __init__(self, git_url):
        self._git_url = git_url
        self._name = git_url.split('/')[-1]
        # print('Project name: ', self._name)
        if self._name.endswith('.git'):
            self._name = self._name[:-4]

        # check if project already cloned
        if os.path.exists(projectDir + '/' + self._name):
            return
        else:
            self._clone(git_url)

    def _clone(self, git_url):
        # clone the git repository to the project directory
        try:
            # if Project folder not exist, create it
            if not os.path.exists(projectDir):
                os.makedirs(projectDir)
            return subprocess.check_output(['git', 'clone', git_url,
  projectDir + '/' + self._name], universal_newlines=True)
        except subprocess.CalledProcessError as e:
            raise Exception(f'Error cloning project: {e}')

    def recognizeProjectFramework(self) -> str:
        # TODO: Implement project framework recognition
        return 'flutter'
        pass

    def _setFramework(self, framework) -> None:
        self._framework = framework

    def getFramework(self) -> str:
        return self._framework

    def getName(self) -> str:
        return self._name

    def getPath(self) -> str:
        return projectDir + '/' + self._name

    def getFileContent(self, fileDir: str) -> str:
        """_summary_

        Args:
```

```
            fileDir (str): file directory relative to project directory

        Returns:
            str: file content
        """
        with open(os.path.join(projectDir, self.getName(), fileDir), 'r') as f:
            return f.read()
```

Listing A.1: Project class.

```
from ProjectManager import Project, projectDir, os, subprocess, sdkDir

sdkDir = os.path.join(sdkDir, 'flutter')

class Flutter(Project): # Inherit from Project class

    def __init__(self, git_url):
        super().__init__(git_url)
        self._setFramework('Flutter')
        self._checkSDK()
        self._flutterPubGet()
        self._addTestDependency()
        self.yaml_name = self._getYamlName()
        # self._createSampleProject('sample')

    def _runFlutterCLI(self, args, isRaiseException=False) -> tuple:
        prjDir = os.path.join(projectDir, self.getName())
        flutterBatDir = os.path.join(sdkDir, 'bin', 'flutter')

        cmd = [flutterBatDir]
        # args handling
        # if args is a string that have space, convert it to list
        if isinstance(args, str) and ' ' in args:
            args = args.split()
        if isinstance(args, list):
            cmd.extend(args)

        # run cmd via subprocess
        try:
            process = subprocess.Popen(cmd, cwd=prjDir, stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=True, encoding='utf-8', shell=True)
            stdout, stderr = process.communicate()
            if process.returncode != 0 and isRaiseException:
                raise Exception(f'Error running flutter command: {stderr}')
            return stdout, stderr
        except subprocess.CalledProcessError as e:
            if isRaiseException:
                raise Exception(f'Error running flutter command: {e}')
            return e.__dict__, e.args

    def _checkSDK(self) -> None:
        # Check if flutter sdk is installed
        if not os.path.exists(sdkDir):
            print('Flutter SDK not found')
            return
        # run sdk from sdkDir
```

```
        try:
            self._runFlutterCLI('--version', isRaiseException=True)
        except subprocess.CalledProcessError as e:
            raise Exception(f'Error checking flutter sdk: {e}')

        # print(result)

    def _getYamlName(self) -> str:

        yamlContent = self.getFileContent('pubspec.yaml')
        # print(yamlContent)
        # first line should define the name of the project: "name: ....."
        return yamlContent.split('\n')[0].split('name: ')[1].strip()


    # function for testing only. Do not use in production
    def _createSampleProject(self, prjName) -> str:
        try:
            # cannot use _runFlutterCLI because no project directory yet
            # result = self._runFlutterCLI(['create', prjName], isRaiseException=True)
            result = subprocess.check_output([os.path.join(sdkDir, 'bin', 'flutter'), 'create', prjName],cwd=projectDir, universal_newlines=True, encoding='utf-8',  shell=True)

        except subprocess.CalledProcessError as e:
            raise Exception(f'Error creating flutter project: {e}')
        return result

    def _flutterPubGet(self) -> None:
        # prjDir = os.path.join(projectDir, self.getName())
        # flutterBatDir = os.path.join(sdkDir, 'bin', 'flutter.bat')

        try:
            # result = subprocess.check_output([flutterBatDir, 'pub', 'get'], cwd=prjDir, universal_newlines=True)
            self._runFlutterCLI(['pub', 'get', '--no-example'], isRaiseException=True)
        except subprocess.CalledProcessError as e:
            raise Exception(f'Error running flutter pub get: {e}')

        # print(result)

    def _addTestDependency(self) -> None:
        # run pub add test
        try:
            self._runFlutterCLI(['pub', 'add', 'test'], isRaiseException=True)
        except subprocess.CalledProcessError as e:
            raise Exception(f'Error adding test dependency: {e}')
        # print(result)

    def create_test(self, filename, content, isOverWrite = False) -> None:
        # create test file in the test directory
        # check if test directory exists
        testDir = os.path.join(projectDir, self.getName(), 'test')
```

```python
        if not os.path.exists(testDir):
            os.makedirs(testDir)
        # check if file exists
        fileDir = os.path.join(testDir, filename)
        if os.path.exists(fileDir) and not isOverWrite:
            raise Exception(f'File {fileDir} already exists')
        # create file
        with open(fileDir, 'w') as f:
            f.write(content)

    def get_test_content(self, filename) -> str:
        # use getFileContent to get the content of the test file
        testDir = os.path.join(projectDir, self.getName(), 'test')
        fileDir = os.path.join(testDir, filename)
        if not os.path.exists(fileDir):
            raise Exception(f'File {fileDir} does not exist')
        return self.getFileContent(fileDir)

    # return tuple (result, error)
    def run_test(self, filename) -> tuple:
        fileDir = os.path.join('test', filename)
        try:
            result = self._runFlutterCLI(['test', fileDir])
        except subprocess.CalledProcessError as e:
            raise Exception(f'Error running flutter test: {e}')
        return result
        pass

    def validate(self) -> str:
        # run all tests in the test directory
        testDir = os.path.join(projectDir, self.getName(), 'test')
        for file in os.listdir(testDir):
            if file.endswith('.dart'):
                result, err = self.run_test(file)
                if err:
                    return err

        return ''

    def getListSourceFiles(self) -> list[str]:
            """_summary_

            Returns:
                list[str]: list of source files in the project relative to project directory
            """
            prjDir = os.path.join(projectDir, self.getName())
            libDir = os.path.join(prjDir, 'lib')
            sourceFiles = []

            # find main.dart first
            if os.path.exists(os.path.join(libDir, 'main.dart')):
                sourceFiles.append(os.path.relpath(os.path.join(libDir, 'main.dart'), prjDir))

            for root, dirs, files in os.walk(libDir):
                for file in files:
                    if file.endswith('.dart') and os.path.relpath(os.path.join(root, file), prjDir) not in sourceFiles:
```

```
                        sourceFiles.append(os.path.relpath(os.path.
join(root, file), prjDir))

            return sourceFiles

    def __str__(self) -> str:
        return f'Flutter project {self.getName()} created from {self.
_git_url}'

    pass

```

Listing A.2: `Flutter` class - subclass of Project.

```
from ProjectManager import Project
from .Flutter import FlutterAnalyzeStrategy
from .AI_Agent import AI_Agent

class DependencyDiagram:

    blocks = []
    connections = []

    def __init__(self, project: Project) -> None:
        self.project = project
        self._generateDiagram()
        self.ai_agent = AI_Agent()
        self._getPredictions()

    def _generateDiagram(self) -> None:
        # Analyze project abstractly to project's framework
        framework = self.project.getFramework()
        functionName = framework + 'AnalyzeStrategy'
        if functionName in globals():
            globals()[functionName](self)
        else:
            raise Exception('Framework not supported')

    def _getPredictions(self) -> None:
        for block in self.blocks:
            block.setPrediction(self.ai_agent.
generate_BLA_prediction(source_code=block.getContentNoComment(),
chat_history=[]))
```

Listing A.3: `DependencyDiagram` class.

```
class Block:
def __init__(self, name: str, content: str, type: str) -> None:
    self.name = name
    self.content = content
    self.type = type

def getContentNoComment(self) -> str:
    # no split by line
    content = self.content
    res = ''
    i = 0
    isCommentSingleLine = False
    isCommentMultiLine = False
    while i < len(self.content)-1:
```

```
            # if \n, reset isCommentSingleLine
            if content[i] == '\n':
                isCommentSingleLine = False
            if content[i] == '/' and content[i+1] == '*':
                isCommentMultiLine = True
            if content[i] == '/' and content[i+1] == '/':
                isCommentSingleLine = True
            if not isCommentSingleLine and not isCommentMultiLine:
                res += content[i]
            if content[i] == '*' and content[i+1] == '/':
                isCommentMultiLine = False
                i+=1
            i+=1

        # delete all empty lines
        res = '\n'.join([line for line in res.split('\n') if line.strip() != ''])

        return res

    def setPrediction(self, prediction: str) -> None:
        self.prediction = prediction

    def getPrediction(self) -> str:
        return self.prediction
```

Listing A.4: `Block` class.

```
class BlockType:
    FILE = 'File'
    CLASS = 'Class'
    ABSTRACT_CLASS = 'AbstractClass'
    ENUM = 'Enum'
    GLOBAL_VAR = 'GlobalVar'
    FUNCTION = 'Function'
    CLASS_CONSTRUCTOR = 'ClassConstructor'
    CLASS_FUNCTION = 'ClassFunction'
    CLASS_ATTRIBUTE = 'ClassAttribute'
```

Listing A.5: `BlockType` class (Enumerate).

```
class Connection:
def __init__(self, head: Block, tail: Block, type: str):
    self.head = head
    self.tail = tail
    self.type = type
```

Listing A.6: `Connection` class.

```
class ConnectionType:
    EXTEND = 'Extend'
    IMPLEMENT = 'Implement'
    CONTAIN = 'Contain'
    EXTEND = 'Extend'
    USE = 'Use'
    CALL = 'Call'
    IMPORT = 'Import'
```

Listing A.7: `ConnectionType` class (Enumerate).

```python
    def FlutterAnalyzeStrategy(diagram) -> None:
    # print('Flutter analyze strategy')
    # print(diagram)
    fileList = diagram.project.getListSourceFiles()
    # print(fileList)
    # create a block for main first
    mainfileDir = fileList[0]
    mainFileContent = diagram.project.getFileContent(mainfileDir)
    # turn \ into /
    mainfileDir = mainfileDir.replace('\\','/')
    # print(mainfileDir)
    mainBlock = Block(mainfileDir, mainFileContent, BlockType.FILE)
    # print(mainBlock)

    diagram.blocks.append(mainBlock)

    ImportAnalyzer(diagram, diagram.blocks[0])

    ContainAnalyzer(diagram, diagram.blocks[0])

    CallAnalyzer(diagram, diagram.blocks[0])
```

Listing A.8: `FlutterAnalyzeStrategy` function.

```python
def ImportAnalyzer(diagram, block):
    currContent = block.content
    currType = block.type
    # print("Current content: ", currentContent)
    # print("Current type: ", currentType)

    # analyze imports
    if (currType == 'File'):
        importLines = [line.strip() for line in currContent.split('\n') if line.strip().startswith('import')]
        # print(importLines)
        blocks = []
        for line in importLines:
            # print(line)
            directory = line.split(' ')[1].replace(';', '')
            # delete first and last character => delete quotes
            directory = directory[1:-1]
            # print(directory)
            # 3 cases: import from other package, import from project, import as relative path
            if directory.startswith('package:'):
                # import from other package, import from project
                prjName = diagram.project.yaml_name
                if directory.startswith(f'package:{prjName}'):
                    # import from project
                    # create block for this file and connection
                    fileDir = directory.split(f'package:{prjName}')[1]
                    fileDir = 'lib' + fileDir
                    fileContent = diagram.project.getFileContent(fileDir)
                    # if fileDir is not in Diagram.blocks
                    if not any(block.name == fileDir for block in diagram.blocks):
                        blocks.append(Block(fileDir, fileContent, BlockType.FILE))
```

```
                    else: diagram.connections.append(Connection(block, [b for b in diagram.blocks if b.name == fileDir][0], ConnectionType.IMPORT))
            else:
                # import as relative path
                currentDir = block.name #ex: lib/main.dart
                currentDir = currentDir.split('/')
                currentDir.pop()
                currentDir = '/'.join(currentDir)
                combineDir = os.path.normpath(os.path.join(currentDir, directory))
                # print(combineDir)
                fileContent = diagram.project.getFileContent(combineDir)
                if combineDir not in [block.name for block in diagram.blocks]:
                    # turn \ into /
                    combineDir = combineDir.replace('\\','/')
                    blocks.append(Block(combineDir, fileContent, BlockType.FILE))
                else: diagram.connections.append(Connection(block, [b for b in diagram.blocks if b.name == combineDir][0], ConnectionType.IMPORT))

        for b in blocks:
            # print(b)
            diagram.blocks.append(b)
            diagram.connections.append(Connection(block, b, ConnectionType.IMPORT))
            ImportAnalyzer(diagram, b)
```

Listing A.9: `ImportAnalyzer` function.

```
def ContainAnalyzer(diagram, block, visited = []):
    visited.append(block)
    currType = block.type

    # print("Current content: ", currContent)
    # print("Current type: ", currType)

    # keep analyze if type is file or class or abstract class
    if (currType == BlockType.FILE
        or currType == BlockType.CLASS
        or currType == BlockType.ABSTRACT_CLASS
        ):
        content = block.getContentNoComment()
        # print(content)
        lines = content.split('\n')
        # if type is file, analyze classes and functions (standalone functions)
        # if type is class, analyze functions
        blocks = []
        # File analyzing
        if (currType == BlockType.FILE):
            # two cases: class and abstract class
            if 'class ' in content:
                # this file have class(es)
                isClassContent = False
                openedBracket = 0
                className = ''
```

```
            classContent = []
            # class or final class
            for line in lines:
                if line.strip().startswith('class ') or line.strip().startswith('final class '):
                    # first line of class
                    # NOTE: there is no class inside class
                    # get class name
                    className = line.split('class ')[1].split('{')[0].strip()
                    # print(className)
                    isClassContent = True
                    classContent.append(line)
                elif '}' in line and isClassContent:
                    # two cases: class end or function end
                    classContent.append(line)
                    if '{' in line:
                        continue
                    if openedBracket > 0:
                        openedBracket = openedBracket - 1
                    else:
                        # class end
                        isClassContent = False
                        classContent = '\n'.join(classContent)
                        blocks.append(Block(className, classContent, BlockType.CLASS))
                        classContent = []
                elif '{' in line and isClassContent:
                    openedBracket = openedBracket + 1
                    classContent.append(line)
                elif isClassContent:
                    classContent.append(line)

            # abstract class
            if 'abstract class ' in content or 'abstract final class ' in content:
                isAbstractClassContent = False
                openedBracket = 0
                className = ''
                classContent = []
                for line in lines:
                    if line.strip().startswith('abstract class ') or line.strip().startswith('abstract final class '):
                        # first line of class
                        # NOTE: there is no class inside class
                        # get class name
                        if line.strip().startswith('abstract class '):

                            className = line.split('abstract class ')[1].split('{')[0].strip()
                        # Check if it's an abstract final class
                        if line.strip().startswith('abstract final class '):
                            className = line.split('abstract final class ')[1].split('{')[0].strip()
                        # print(className)
                        isAbstractClassContent = True
                        classContent.append(line)
```

```
                    elif '}' in line and isAbstractClassContent:
                        # two cases: class end or function end
                        classContent.append(line)
                        if '{' in line:
                            continue
                        if openedBracket > 0:
                            openedBracket = openedBracket - 1
                        else:
                            # class end
                            isAbstractClassContent = False
                            classContent = '\n'.join(classContent)
                            blocks.append(Block(className, classContent, BlockType.ABSTRACT_CLASS))
                            classContent = []
                    elif '{' in line and isAbstractClassContent:
                        openedBracket = openedBracket + 1
                        classContent.append(line)
                    elif isAbstractClassContent:
                        classContent.append(line)

        # enum
        if 'enum ' in content:
            # no {} in enum
            # maybe () in enum
            # once } is found, enum end
            isEnumContent = False
            enumName = ''
            enumContent = []
            for line in lines:
                if line.strip().startswith('enum '):
                    # first line of enum
                    enumName = line.split('enum ')[1].split('{')[0].strip()
                    isEnumContent = True
                    enumContent.append(line)
                elif '}' in line and isEnumContent:
                    enumContent.append(line)
                    isEnumContent = False
                    enumContent = '\n'.join(enumContent)
                    blocks.append(Block(enumName, enumContent, BlockType.ENUM))
                    # print(enumContent)
                elif isEnumContent:
                    enumContent.append(line)

        # function
        # standalone function / GlobalVar only!
        # strat: get rid of all analyzed class and enum first
        leftoverContent = content
        for b in blocks:
            leftoverContent = leftoverContent.replace(b.content, '')
        # get rid of import line
        leftoverContent = '\n'.join([line for line in leftoverContent.split('\n') if not line.strip().startswith('import')])
        # remove empty lines
        leftoverContent = '\n'.join([line for line in leftoverContent.split('\n') if line.strip() != ''])
```

```python
        # print("========Leftover content========")
        # print(block.name)
        # print(leftoverContent)

        # two case of function: difined return type or not (dynamic)
        # variable must have a type
        # print("========Function and GlobalVar========")
        funcAndVarBlocks = extract_functions_and_globals(leftoverContent)
        blocks.extend(funcAndVarBlocks)

    # Class analyzing
    if (currType in (BlockType.CLASS)):
        # two cases: class function and class attribute
        content = block.getContentNoComment() # should be no difference between content and contentNoComment
        # print(content)
        classContentBlock = extract_class_content(content)
        blocks.extend(classContentBlock)

    # blocks recursive
    for b in blocks:
        # print(b)
        # print(b.content)
        diagram.blocks.append(b)
        diagram.connections.append(Connection(block, b, ConnectionType.CONTAIN))
        ContainAnalyzer(diagram, b, visited=visited)

# find connection connected to this block and not visited
connectedBlocks = [c.tail for c in diagram.connections if c.head == block and c.tail not in visited]
for b in connectedBlocks:
    ContainAnalyzer(diagram, b, visited=visited)
```

Listing A.10: `ContainAnalyzer` function.

```python
def CallAnalyzer(diagram, block, visited = []):
if block in visited:
    return

visited.append(block)
currType = block.type

# NOTE strat: 2-layer recursive
if currType in (BlockType.FILE):
    # find connected file (imported file)
    connectedFiles = [conn.tail for conn in diagram.connections if conn.head == block and conn.type == ConnectionType.IMPORT]

    for file in connectedFiles:
        # print("Imported file:")
        # print(file)
        # find all class/function/variable in file. Avoid BlockType.FILE
        connectedBlocks = [conn.tail for conn in diagram.connections if conn.head == file and conn.type == ConnectionType.CONTAIN]
```

```
            currentBlocks = [conn.tail for conn in diagram.connections if conn.head == block and conn.type == ConnectionType.CONTAIN]


            _CallAnalyzer(diagram, currentBlocks, connectedBlocks, visited)
            # import based recursive call
            CallAnalyzer(diagram, file, visited)

def _CallAnalyzer(diagram, thisFile, callables, visited=[]):
    # thisFile: blocks of contains in current file
    # callables: blocks of contains in imported file
    callables.extend(thisFile)

    # printStuff(thisFile, callables)

    for block in thisFile:
        if block in visited:
            continue

        if block.type in (BlockType.ABSTRACT_CLASS, BlockType.CLASS):
            # extend connection analyze
            name = block.name
            # print(name)
            # first word is class name
            classname = name.split()[0]
            otherInfo = name[len(classname):]
            for connBlock in callables:
                if connBlock.type in (BlockType.ABSTRACT_CLASS, BlockType.CLASS):
                    className = connBlock.name.split()[0]
                    # if className found in otherInfo, create a connection ConnectionType.EXTEND
                    if className in otherInfo:
                        diagram.connections.append(Connection(block, connBlock, ConnectionType.EXTEND))
                        # print(f"Extend connection: {block} --> {connBlock}")

            # split class, abstract class
            innerBlocks = [conn.tail for conn in diagram.connections if conn.head == block and conn.type == ConnectionType.CONTAIN]
            # magic recursive calls at 4 a.m
            visited.append(block)
            _CallAnalyzer(diagram, innerBlocks, callables, visited)


            continue
        else:
            visited.append(block)
            # analyze calls in block
            # If called, create a connection ConnectionType.CALL

            fullcontent = block.getContentNoComment()
            # print("=====================================")
            # print(block)
            # print(fullcontent)

            content = ''
```

```
            # Extract content only, exclude function name, params
            if block.type in (BlockType.FUNCTION, BlockType.CLASS_FUNCTION):
                if '=>' in fullcontent:
                    # take content from => to ;
                    # add a ; to the end of content
                    fullcontent = fullcontent + ';'
                    content = fullcontent[fullcontent.index('=>')+2:]
                    content = content[:content.index(';') + 1]

                else:
                    roundbracketOpened = 0
                    initialRoundBracket = False
                    curlybracketOpened = 0
                    isContent = False
                    for char in fullcontent:
                        # params section
                        if char == '(' and not isContent and not initialRoundBracket:
                            initialRoundBracket = True
                            roundbracketOpened += 1
                        if char == ')' and not isContent:
                            roundbracketOpened -= 1
                            if roundbracketOpened == 0 and initialRoundBracket:
                                initialRoundBracket = False
                        if char == '{':
                            if not isContent and not initialRoundBracket:
                                isContent = True
                            curlybracketOpened += 1
                        if char == '}' and isContent:
                            curlybracketOpened -= 1
                            if curlybracketOpened == 0:
                                isContent = False
                                content += char
                                break
                        if isContent:
                            content += char
            if block.type in (BlockType.CLASS_ATTRIBUTE):
                # extract content from = to ;
                # add a ; to the end of content
                fullcontent = fullcontent + ';'
                content = fullcontent[fullcontent.index('=')+1:]
                content = content[:content.index(';') + 1]
            # print("===================Block=====================")
            # print("Block name: ", block.name)
            # print("===================Extracted content ====================")
            # print(content)
            # print("=========================================================")
            # printStuff(thisFile, callables)
            callablesName = getCallablesName(callables)
            for name, connBlock in callablesName:
                # print(f"Name: {name}, Block name: {connBlock.name}")
                # find name in content
                # name found can be next to any non-word character or start of line and end of line
```

```python
                regex = re.compile(r'(?<![a-zA-Z0-9_])' + re.escape(name) + r'(?![a-zA-Z0-9_])')
                if regex.search(content):
                    # check if connection already exists
                    if not any(conn.head == block and conn.tail == connBlock and conn.type == ConnectionType.CALL for conn in diagram.connections):
                        diagram.connections.append(Connection(block, connBlock, ConnectionType.CALL))
                        # print(f"Call connection: {block} --> {connBlock}")
```

Listing A.11: `CallAnalyzer` function.

```python
class AI_Agent:
    def __init__(self) -> None:
        if load_dotenv(override=True) == False:
            raise Exception("Failed to load .env file")
        base_url = os.getenv('BASE_URL')
        model_name = os.getenv('BLA_LLM_MODEL')
        embed_model = os.getenv('EMBED_MODEL')
        self.model = ChatOpenAI(base_url=base_url, model=model_name, temperature=0)
        self.embeddings = OpenAIEmbeddings(
            base_url=base_url,
            model=embed_model,
            # critical for LM studio mod
            check_embedding_ctx_length=False
        )
        # load vector store
        self.store_names = {
            # "dart_programming_tutorial": "dart_programming_tutorial.pdf",
            # "DartLangSpecDraft": "DartLangSpecDraft.pdf",
            "flutter_tutorial": "flutter_tutorial.pdf",
        }
        for store_name, doc_name in self.store_names.items():
            if not self._check_if_vector_store_exists(store_name):
                docs = self._load_document(doc_name)
                chunks = self._split_document(docs)
                self._create_vector_store(chunks, store_name)

        dbs = []
        for store_name in self.store_names:
            dbs.append(
                Chroma(persist_directory=os.path.join(db_dir, store_name),
                       embedding_function=self.embeddings)
            )
        self.retrievers = []
        for db in dbs:
            self.retrievers.append(
                db.as_retriever(
                    # search_type="similarity",
                    # search_kwargs={"k": docs_num},
                    # search_type="mmr",
                    # search_kwargs={"k": docs_num, "fetch_k": 20, "lambda_mult": 0.5}
                    search_type="similarity_score_threshold",
                    search_kwargs={
```

```
                        'score_threshold': 0.4,
                        'k': 1,
                    }
                )
            )
        self._agent_init()
    def generate_BLA_prediction(
        self,
        source_code: str,
        chat_history: list
    ) -> str:
        # First use the agent to analyze the code
        response = self.agent_executor.invoke(
            {
                "input": source_code,
                "chat_history": chat_history,
            }
        )

        # Then use a direct call to the LLM to structure the output properly
        structured_prompt = (
            "Based on the following analysis of code, create a structured response with the following sections:\n"
            "1. Brief explanation of what the code does\n"
            "2. Testability assessment\n"
            "3. TESTING SCENARIOS in the exact format shown below:\n\n"
            "TESTING SCENARIOS:\n"
            "1. [Descriptive Test Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n"
            "2. [Descriptive Test Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n"
            "3. [Descriptive Test Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n\n"
            "For test names, use descriptive names that clearly indicate the purpose of the test, such as:\n"
            "- 'ValidPalindromeCheck' instead of 'Scenario Name'\n"
            "- 'EmptyStringHandling' instead of generic names\n"
            "- 'BoundaryConditionTest' for edge cases\n"
            "- 'SpecialCharactersTest' for specific input types\n\n"
            "Include at least 4-5 different test scenarios covering normal cases, edge cases, and special conditions.\n"
            "Analysis to structure: " + response["output"]
        )

        structured_response = self.model.invoke(structured_prompt)
        return structured_response.content



    def _agent_init(self) -> None:
        contextualize_q_system_prompt = (
            "Given a chat history, user request and the latest piece of user source code, "
            "which might reference context in the chat history, "
```

```
            "formulate a statement that can be used to query the model for useful reference."
            "Do NOT include the user request in the query."
            # "DO NOT add the sentence 'Without more context or specific questions about the code, I can't provide a more detailed explanation' in the answer."
        )
        contextualize_q_prompt = ChatPromptTemplate.from_messages(
            [
                ("system", contextualize_q_system_prompt),
                MessagesPlaceholder("chat_history"),
                ("human", "{input}"),
            ]
        )
        # Create a history-aware retriever
        # This uses the LLM to help reformulate the question based on chat history
        history_aware_retrievers = []

        for retriever in self.retrievers:
            history_aware_retrievers.append(
                create_history_aware_retriever(
                    self.model, retriever, contextualize_q_prompt
                )
            )

        bla_system_prompt = (
            "You are an AI assistant that analyzes Flutter/Dart source code to identify its business logic for test generation.\n"
            "You can provide helpful answers using available tools.\n"
            "For the given code snippet:\n\n"
            "1. FUNCTION ANALYSIS:\n"
            "   - What is the purpose of this function/class?\n"
            "   - What are the inputs (parameters) and their types?\n"
            "   - What is the expected output (return value) and its type?\n"
            "   - What algorithm or logic does it implement?\n\n"
            "2. TESTABILITY ASSESSMENT:\n"
            "   - Can this code be tested? If yes, what type of test is appropriate (unit/widget/integration)?\n"
            "   - Are there any dependencies that might complicate testing?\n\n"
            "3. TESTING SCENARIOS:\n"
            "   ALWAYS include at least 3-5 specific test scenarios using EXACTLY this format:\n\n"
            "   TESTING SCENARIOS:\n"
            "   1. [Scenario Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n"
            "   2. [Scenario Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n"
            "   3. [Scenario Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n\n"
            "Keep your analysis concise but precise. DO NOT include the source code in your answer.\n"
            "The TESTING SCENARIOS section MUST follow the exact format shown above, with specific input values and expected
```

```
outputs.\n"
            "If the code's purpose is unclear, make your best inference based on the implementation details.\n"
            "{context}"
        )

        bla_prompt = ChatPromptTemplate.from_messages(
            [
                ("system", bla_system_prompt),
                MessagesPlaceholder("chat_history"),
                ("human", "{input}"),
            ]
        )
        bla_chain = create_stuff_documents_chain(self.model, bla_prompt)

        rag_chains = []
        for retriever in history_aware_retrievers:
            rag_chains.append(create_retrieval_chain(retriever, bla_chain))

        react_docstore_prompt = hub.pull("hwchase17/react")

        tools = []

        store_names = []
        for store_name, doc_name in self.store_names.items():
            store_names.append(store_name)

        for i in range(len(store_names)):
            # print(f"{store_names[i]}")
            tools.append(
                Tool(
                    name=f"Get code explaination from {store_names[i]}",
                    func=lambda input, **kwargs: rag_chains[i].invoke(
                        {
                            "input": input,
                            "chat_history": kwargs.get("chat_history", [])
                        }
                    ),
                    description=f"Retrieve documents from the vector store {store_names[i]}",
                )
            )
        agent = create_react_agent(
            llm=self.model,
            tools=tools,
            prompt=react_docstore_prompt,
        )

        self.agent_executor = AgentExecutor.from_agent_and_tools(
            agent=agent,
            tools=tools,
            handle_parsing_errors=True,
            verbose=True,
        )
        pass
```

```python
    # Function to create and persist vector store
    def _create_vector_store(self, docs, store_name, is_overwrite=False) -> None:
        persistent_directory = os.path.join(db_dir, store_name)
        # delete the directory if it exists and needed
        if is_overwrite and os.path.exists(persistent_directory):
            shutil.rmtree(persistent_directory) # remove the directory

        if not os.path.exists(persistent_directory):
            print(f"\n--- Creating vector store {store_name} ---")
            db = Chroma.from_documents(
                docs, self.embeddings, persist_directory=persistent_directory
            )
            print(f"--- Finished creating vector store {store_name} ---")
        else:
            print(
                f"Vector store {store_name} already exists. No need to initialize.")
    def _load_document(self, doc_name):
        file_path = os.path.join(docs_dir, doc_name)
        if not os.path.exists(file_path):
            raise FileNotFoundError(
                f"_load_document: The file {file_path} does not exist. Please check the path."
            )
        file_extension = os.path.splitext(file_path)[1]
        # check if the file extension is supported
        if file_extension not in file_loader_map:
            raise Exception(f"_load_document: Unsupported file extension: {file_extension} for file: {file_path}")
        loader = PyPDFLoader(file_path=file_path)
        return loader.load()

    def _split_document(self, documents, chunk_size=1000, chunk_overlap=100):
        text_splitter = SentenceTransformersTokenTextSplitter(
            chunk_size=chunk_size, chunk_overlap=chunk_overlap
        )
        return text_splitter.split_documents(documents)

    def _check_if_vector_store_exists(self, store_name) -> bool:
        persistent_directory = os.path.join(db_dir, store_name)
        return os.path.exists(persistent_directory)
```

Listing A.12: AI_Agent class.

```
OPENAI_API_KEY=sk-this-key-is-just-a-placeholder
LANGCHAIN_API_KEY=sk-this-key-is-just-a-placeholder
LANGCHAIN_PROJECT=TestGenie

BASE_URL=
BLA_LLM_MODEL=
TG_LLM_MODEL=

EMBED_MODEL=
```

Listing A.13: Sample .env file.

```python
class Test_Generator:
def __init__(self) -> None:
    if load_dotenv(override=True) == False:
        raise Exception("Failed to load .env file")
    base_url = os.getenv('BASE_URL')
    model_name = os.getenv('TG_LLM_MODEL')
    embed_model = os.getenv('EMBED_MODEL')
    self.model = ChatOpenAI(base_url=base_url, model=model_name, temperature=0) # type: ignore
    self.embeddings = OpenAIEmbeddings(
        base_url=base_url,
        model=embed_model, # type: ignore
        # critical for LM studio mod
        check_embedding_ctx_length=False
    )
    # load vector store process
    self.store_names = self._getStoreList()
    for store_name, doc_name in self.store_names.items():
        if not self._check_if_vector_store_exists(store_name):
            docs = self._load_document(doc_name)
            chunks = self._split_document(docs)
            self._create_vector_store(chunks, store_name)

    self.dbs = []
    for store_name in self.store_names.keys():
        self.dbs.append(Chroma(persist_directory=os.path.join(db_dir, store_name), embedding_function=self.embeddings))

    self.retrievers = []
    for db in self.dbs:
        self.retrievers.append(
            db.as_retriever(
                search_type="similarity_score_threshold",
                search_kwargs={
                    'score_threshold': 0.2,
                    'k': 1,
                }
            )
        )

    # Create error cache to avoid repeating fixes
    self.error_fix_cache = {}

    # Set of attempted fixes for error tracking
    self.attempted_fixes_for_error = {}

    # Maximum retries for a single error
    self.max_error_fix_attempts = 3

def generate_test_case(
    self,
    package_name: str,
    code_location: str,
    function_name_and_arguments: str,
    prediction: str,
) -> str:
    """
    Generate a test case for a function based on the prediction and code details.
```

```

        Args:
            package_name: Name of the package (for import statements)
            code_location: Location of the code file to test (path within the package)
            function_name_and_arguments: Function signature with arguments
            prediction: Description of what the function does

        Returns:
            Generated test case as a string (clean Dart source code only)
        """
        try:
        # Extract the structured sections from prediction if needed
        # Check if prediction contains the expected structure
            if "TESTING SCENARIOS:" not in prediction and "Brief" not in prediction:
                # If prediction isn't properly structured, try to structure it
                structured_prompt = (
                    "Structure this analysis into the following format:\n"
                    "1. Brief explanation of what the code does\n"
                    "2. Testability assessment\n"
                    "3. TESTING SCENARIOS in this exact format:\n\n"
                    "TESTING SCENARIOS:\n"
                    "1. [Descriptive Test Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n"
                    "2. [Descriptive Test Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n"
                    "3. [Descriptive Test Name]: Verify that [functionality]. Input: [specific input values]. Expected: [specific output/behavior].\n\n"
                    "Analysis to structure: " + prediction
                )

                structured_response = self.model.invoke(structured_prompt)
                prediction = structured_response.content

            # Generate the test case with the structured prediction
            raw_output = self._generate_clean_test(package_name, code_location, function_name_and_arguments, prediction)

            # Clean up any markdown formatting that might be present
            cleaned_output = self._clean_code_output(raw_output)

            return cleaned_output
        except Exception as e:
            print(f"Error generating test case: {str(e)}")
            return f"// Error generating test case: {str(e)}"

    def fix_generated_code(
        self,
        error_message: str,
        current_test_code: str,
```

```python
        prediction: str,
    ) -> str:
        """
        Fix issues in generated test code based on error messages from the Dart SDK.
        Enhanced with online search and error pattern learning capabilities.

        Args:
            error_message: The error message from the Dart SDK
            current_test_code: The current test code that has issues
            prediction: The original prediction about what the function does

        Returns:
            Fixed test code that addresses the errors
        """
        try:
            # Create a unique identifier for this error+code combination to track fix attempts
            error_hash = self._generate_error_hash(error_message, current_test_code)

            # Check if we've seen and fixed this exact error before
            if error_hash in self.error_fix_cache:
                print(f"Using cached fix for error: {error_hash[:10]}...")
                return self.error_fix_cache[error_hash]

            # Track fix attempts to avoid infinite loops
            if error_hash not in self.attempted_fixes_for_error:
                self.attempted_fixes_for_error[error_hash] = 0

            self.attempted_fixes_for_error[error_hash] += 1

            # If we've tried too many times, use different strategies or bail out
            if self.attempted_fixes_for_error[error_hash] > self.max_error_fix_attempts:
                print(f"Maximum fix attempts reached for error {error_hash[:10]}. Applying emergency fix...")
                # Apply emergency fix that attempts to produce at least a basic test case
                emergency_fixed = self._emergency_fix(current_test_code, error_message)
                self.error_fix_cache[error_hash] = emergency_fixed
                return emergency_fixed

            # Extract unique errors from the potentially repetitive error message
            unique_errors = self._extract_unique_errors(error_message)

            # 1. First try our standard approach
            if self.attempted_fixes_for_error[error_hash] == 1:
                fixed_code = self._standard_ai_fix(unique_errors, current_test_code)

            # 2. If that didn't work, search online for solutions
            elif self.attempted_fixes_for_error[error_hash] == 2:
```

```python
                # Search for online solutions for this error
                online_solutions = self._search_for_error_solutions(unique_errors)

                # Use online solutions to enhance fix prompt
                fixed_code = self._ai_fix_with_online_knowledge(unique_errors, current_test_code, online_solutions)

            # 3. Final attempt with different approach
            else:
                fixed_code = self._comprehensive_repair_attempt(error_message, current_test_code, prediction)

            # Apply additional specific rule-based fixes
            fixed_code = self._apply_specific_fixes(fixed_code, unique_errors)

            # Cache the successful fix for this error
            self.error_fix_cache[error_hash] = fixed_code

            return fixed_code

        except Exception as e:
            print(f"Error fixing test code: {str(e)}")
            # Try a simpler approach with manual fixes for common errors
            try:
                manually_fixed = self._manual_fix_common_errors(current_test_code, error_message)
                return manually_fixed
            except:
                # If all else fails, return the original with error comments
                return f"// Error while trying to fix the code: {str(e)}\n// Original error message: {error_message}\n\n{current_test_code}"
```

Listing A.14: Test_Generator class.

```python
frameworkMap = {
    'flutter': Flutter
}

def getDBMS(git_url) -> DBMS:
    project = Project(git_url)
    framework = project.recognizeProjectFramework()

    if framework in frameworkMap:
        project = frameworkMap[framework](git_url)

    dbms = DBMS(project)

    return dbms

app = Flask(__name__)
CORS(app)

# Post git project url
@app.route('/createProject', methods=['POST'])
def createProject():
```

```
        if not request.json or not 'git_url' in request.json:
            return jsonify({'message': 'Invalid request'})
        git_url = request.json['git_url']
        project = Project(git_url)
        # print(project)
        return jsonify({'message': f'{project}'})

    @app.route('/getDiagram', methods=['POST'])
    def getDiagram():
        # print(request.json)
        if not request.json or not 'git_url' in request.json:
            return jsonify({'message': 'Invalid request'})
        git_url = request.json['git_url']

        dbms = getDBMS(git_url)

        diagram = dbms.getJsonDiagram()
        return jsonify(diagram)

    @app.route('/getDiagram', methods=['OPTIONS'])
    def getDiagramOptions():
        print(request.json)
        print("wrong method")
        return jsonify({'message': 'Options request'})

    @app.route('/getBlockContent', methods=['POST'])
    def getBlockContent():
        if not request.json or not 'git_url' in request.json or not 'block_id' in request.json:
            return jsonify({'message': 'Invalid request'})
        git_url = request.json['git_url']
        blockId = request.json['block_id']

        dbms = getDBMS(git_url)
        blockContent = dbms.getBlockContent(blockId)
        return jsonify(blockContent)

    @app.route('/getBlockPrediction', methods=['POST'])
    def getBlockPrediction():
        if not request.json or not 'git_url' in request.json or not 'block_id' in request.json:
            return jsonify({'message': 'Invalid request'})
        git_url = request.json['git_url']
        blockId = request.json['block_id']

        dbms = getDBMS(git_url)
        blockPrediction = dbms.getBlockPrediction(blockId)
        return jsonify(blockPrediction)

    @app.route('/getBlockDetail', methods=['POST'])
    def getBlockDetail():
        if not request.json or not 'git_url' in request.json or not 'block_id' in request.json:
            return jsonify({'message': 'Invalid request'})
        git_url = request.json['git_url']
        blockId = request.json['block_id']

        dbms = getDBMS(git_url)
        # {
```

```
            # 'content': blockContent,
              # 'prediction': blockPrediction,
        # }
        content = dbms.getBlockContent(blockId)
        prediction = dbms.getBlockPrediction(blockId)
        try:
            test_file_content = dbms.project.get_test_content ('block_' + str(blockId) + '_test.dart')
        except Exception as e:
            test_file_content = ''

        return jsonify({
            'content': content,
            'prediction': prediction,
            'test_file_content': test_file_content,
        })

    # dont know why this is needed
    @app.route('/getBlockDetail', methods=['OPTIONS'])
    def getBlockDetailOptions():
        print(request.json)
        print("wrong method")
        return jsonify({'message': 'Options request'})

    @app.route('/updateBlockPrediction', methods=['POST'])
    def updateBlockPrediction():
        if not request.json or not 'git_url' in request.json or not 'block_id' in request.json or not 'prediction' in request.json:
            return jsonify({'message': 'Invalid request'})
        git_url = request.json['git_url']
        blockId = request.json['block_id']
        prediction = request.json['prediction']

        dbms = getDBMS(git_url)
        dbms.updateBlockPrediction(blockId, prediction)

        return jsonify(
            {
                'message': 'Update success!',
                'code': 200,
                'success': True,
            }
        )

    @app.route('/updateBlockPrediction', methods=['OPTIONS'])
    def updateBlockPredictionOptions():
        print(request.json)
        print("wrong method")
        return jsonify({'message': 'Options request'})

    @app.route('/generateTest', methods=['POST'])
    def generateTest():
        try:
            if not request.json or not 'git_url' in request.json or not 'block_id' in request.json:
                return jsonify({'message': 'Invalid request'})
            git_url = request.json['git_url']
            blockId = request.json['block_id']

```

```
        dbms = getDBMS(git_url)
        tg = Test_Generator()

        testFileContent = tg.generate_test_case(
            package_name= dbms.project.getName(),
            code_location=dbms.getBlockOriginalFile(blockId),
            function_name_and_arguments=dbms.getBlockName(blockId),
            prediction=dbms.getBlockPrediction(blockId),
        )
        file_name = 'block_' + str(blockId) + '_test.dart'

        dbms.project.create_test(
            filename=file_name,
            content=testFileContent,
            isOverWrite=True
        )
        # validation process
        run_result, run_error = dbms.project.run_test(file_name)
        iteration_limit = 5
        while run_error != '' and iteration_limit > 0:
            new_test_content = tg.fix_generated_code(
                error_message=run_error,
                code_location=dbms.getBlockOriginalFile(blockId),
                function_name_and_arguments=dbms.getBlockName(blockId),
                prediction=dbms.getBlockPrediction(blockId),
            )
            dbms.project.create_test(
                filename=file_name,
                content=new_test_content,
                isOverWrite=True
            )
            run_result, run_error = dbms.project.run_test(file_name)
            iteration_limit -= 1

        return jsonify(
            {
                'message': 'Test generation success!',
                'code': 200,
                'success': True,
                'test_file_content': testFileContent,
            }
        )
    except Exception as e:
        return jsonify({'message': str(e)})

@app.route('/generateTest', methods=['OPTIONS'])
def generateTestOptions():
    print(request.json)
    print("wrong method")
    return jsonify({'message': 'Options request'})

if __name__ == '__main__':
    app.run(debug=True)
```

Listing A.15: `main.py` file.

```
class Table:
```

```
    def __init__(self, name: str, columns: dict):
        self.name = name
        self.columns = columns


    def getCreateSQL(self):
        sql = f'CREATE TABLE IF NOT EXISTS {self.name} ('
        for column in self.columns:
            sql += f'{column} {self.columns[column]}, '
        sql = sql[:-2] + ')'
        return sql

    def getSelectSQL(self,fields: list ,conditions: dict):
        # if conditions is empty, return all
        res = f'SELECT '
        if len(fields) == 0:
            res += '*'
        else:
            for field in fields:
                res += f'{field}, '
            res = res[:-2]
        res += f' FROM {self.name}'

        if len(conditions) > 0:
            res += ' WHERE '
            for condition in conditions:
                res += f"{condition} = '{conditions[condition]}' AND "
            res = res[:-4]

        return res
        pass

    def getInsertSQL(self, values: dict):
        sql = f'INSERT INTO {self.name} ('
        for column in values:
            sql += f'{column}, '
        sql = sql[:-2] + ") VALUES ("
        for column in values:
            sql += f"'{values[column]}', "
        sql = sql[:-2] + ')'
        return sql

    def getUpdateSQL(self, values: dict, conditions: dict):
        sql = f"UPDATE {self.name} SET "
        for column in values:
            sql += f"{column} = \'{values[column]}\', "
        sql = sql[:-2] + " WHERE "
        for column in conditions:
            sql += f"{column} = \'{conditions[column]}\' AND "
        sql = sql[:-4]
        return sql
```

Listing A.16: `Table` class.

```
    @staticmethod
    def getTable():
        from DBMS.Table import Table
        return Table(
            'BlockType',
            {
```

```
                'id': 'INT AUTO_INCREMENT PRIMARY KEY',
                'name': 'VARCHAR(255)'
            }
        )
```

Listing A.17: `getTable` function - BlockType class.

```
    @staticmethod
    def getTable():
        from DBMS.Table import Table
        return Table(
            'Block',
            {
                'id': 'INT AUTO_INCREMENT PRIMARY KEY',
                'name': 'VARCHAR(255)',
                'content': 'TEXT',
                'prediction': 'TEXT',
                'type': 'INT',
                '': 'FOREIGN KEY (type) REFERENCES BlockType(id)'
            }
        )
```

Listing A.18: `getTable` function - Block class.

```
    @staticmethod
    def getTable():
        from DBMS.Table import Table
        return Table(
            'ConnectionType',
            {
                'id': 'INT AUTO_INCREMENT PRIMARY KEY',
                'name': 'VARCHAR(255)'
            }
        )
```

Listing A.19: `getTable` function - ConnectionType class.

```
    @staticmethod
    def getTable():
        from DBMS.Table import Table
        return Table(
            'Connection',
            {
                'id': 'INT AUTO_INCREMENT PRIMARY KEY',
                'head': 'INT',
                'tail': 'INT',
                'type': 'INT',
                '': 'FOREIGN KEY (head) REFERENCES Block(id)',
                '':'FOREIGN KEY (tail) REFERENCES Block(id)',
                '': 'FOREIGN KEY (type) REFERENCES ConnectionType(id)'
            }
        )
```

Listing A.20: `getTable` function - Connection class.

```
class DBMS:

    _numberOfTables = 5

```

```
    def __init__(self, project) -> None:
        self.project = project

        # print(self._isDBinit())
        if not self._isDBinit():
            self._initDB()

        # print(self._isProjectExistInDB())
        if not self._isProjectExistInDB():
            self._insertProject()

        else:
            # TODO: do something if project already exist
            pass

    def getJsonDiagram(self) -> dict:
        """
        Get the diagram in json format
        Dict structure:
        {
            project: "project_name",
            blocks: [
                {
                    id: 1,
                    name: "block_name",
                    content: "block_content",
                    prediction: "block_prediction",
                    type: "block_type"
                }
            ]
            connections: [
                {
                    head: 1,
                    tail: 2,
                    type: "connection_type"
                }
            ]
        }
        """
        # fetch diagram from db
        blockQuery = Block.getTable().getSelectSQL(fields=['id','name'
,'type'],conditions={})
        blocks = self.execute(blockQuery)

        connectionQuery = Connection.getTable().getSelectSQL(fields
=[], conditions={})
        connections = self.execute(connectionQuery)

        # print(blocks)
        # print(connections)

        res = {
            'project': self.project.getName(),
            'blocks': [],
            'connections': []
        }
        for block in blocks:
            res['blocks'].append({
                'id': block[0],
```

```
                'name': block[1],
                'type': self._getEnumName('BlockType', block[2]),
            })

        for connection in connections:
            res['connections'].append({
                'head': connection[1],
                'tail': connection[2],
                'type': self._getEnumName('ConnectionType', connection[3])
            })

        return res

        pass

    def getBlockName(self, blockId: int) -> str:
        query = Block.getTable().getSelectSQL(fields=['name'], conditions={
            'id': blockId
        })
        res = self.execute(query)
        return res[0][0]

    def getBlockContent(self, blockId: int) -> str:
        query = Block.getTable().getSelectSQL(fields=['content'], conditions={
            'id': blockId
        })
        res = self.execute(query)
        return res[0][0]

    def getBlockPrediction(self, blockId: int) -> str:
        query = Block.getTable().getSelectSQL(fields=['prediction'], conditions={
            'id': blockId
        })
        res = self.execute(query)
        return res[0][0]

    def getBlockOriginalFile(self, blockId: int) -> str:
        # take blockId as tail, query connection table to get head
        # backtracking until reach FILE block type and return the blockname
        # print('blockId:', blockId)
        query = Connection.getTable().getSelectSQL(fields=['head'], conditions={
            'tail': blockId
        })
        res = self.execute(query)
        if len(res) > 0:
            headId = res[0][0]
            query = Block.getTable().getSelectSQL(fields=['name', 'type'], conditions={
                'id': headId
            })
            res = self.execute(query)
            if len(res) > 0:
                blockType = self._getEnumName('BlockType', res[0][1])
```

```python
                if blockType == 'File':
                    originalFile = res[0][0]
                    # exclude lib/
                    originalFile = originalFile.split('lib/')[1]
                    return originalFile
                else:
                    return self.getBlockOriginalFile(headId)
        pass

    def updateBlockPrediction(self, blockId: int, prediction: str) ->
None:
        query = Block.getTable().getUpdateSQL(
            values={
                'prediction': self._handldApostropheString(prediction)
            },
            conditions={
                'id': blockId
            }
        )
        self.execute(query)

        pass

    def _connect(self):
        self.connection = mysql.connector.connect(
            host='localhost',
            user='root',
            password='1234',
            database='test_genie'
        )
        self.cursor = self.connection.cursor(buffered=True)

    def _close(self):
        self.cursor.close()
        self.connection.close()

    def _isDBinit(self):
        query = 'SHOW TABLES'
        res = self.execute(query)
        return len(res) >= self._numberOfTables

    def execute(self, query) -> list:
        self._connect()

        if type(query) == str:
            self.cursor.execute(query)
        else:
            for q in query:
                self.cursor.execute(q)
        self.connection.commit()

        self._close()
        return self.cursor.fetchall() # type: ignore

    def _initDB(self):
        projectQuery = self.project.getTable().getCreateSQL()
        self.execute(projectQuery)

        blockCreateQuery = Block.getTable().getCreateSQL()
```

```
        self.execute(blockCreateQuery)

        connectionQuery = Connection.getTable().getCreateSQL()
        self.execute(connectionQuery)
        self._insertEnumDB()

    def _insertEnumDB(self):

        blockTypeCreateQuery = BlockType.getTable().getCreateSQL()
        # print(blockTypeQuery)
        self.execute(blockTypeCreateQuery)

        blockTypeInsertQuery = BlockType.getInsertQuery()
        # print(blockTypeInsertQuery)
        self.execute(blockTypeInsertQuery)

        connectionTypeCreateQuery = ConnectionType.getTable().getCreateSQL()
        self.execute(connectionTypeCreateQuery)

        connectionTypeInsertQuery = ConnectionType.getInsertQuery()
        # print(connectionTypeInsertQuery)
        self.execute(connectionTypeInsertQuery)

    def _isProjectExistInDB(self):
        query = self.project.getTable().getSelectSQL(fields=['name'], conditions={
            'name': self.project.getName()
        })
        # print(query)
        res = self.execute(query)
        return len(res) > 0

    def _insertProject(self):
        # print('Inserting project')
        # project table insert
        query = self.project.getTable().getInsertSQL({
            'name': self.project.getName(),
            'directory': self.project.getPath()
        })
        # print(query)
        self.execute(query)
        # diagram insert
        diagram = DependencyDiagram(self.project)
        # not sure if this is needed
        self.diagram = diagram

        blocks = diagram.blocks
        connections = diagram.connections
        self._mapBlocksIntoDB(blocks)
        self._mapConnectionsIntoDB(connections)

    def _mapBlocksIntoDB(self, blocks: list):
        for block in blocks:
            # TODO: handle apostrophe in content
            # map into block table
            query = Block.getTable().getInsertSQL({
                'name': self._handldApostropheString(block.name),
                'content': self._handldApostropheString(block.content)
```

```
,
                'prediction': self._handldApostropheString(block.prediction),
                'type': self._getEnumId('BlockType', block.type)
            })
            self.execute(query)
        pass
    def _mapConnectionsIntoDB(self, connections: list):
        for connection in connections:
            # map into connection table
            query = Connection.getTable().getInsertSQL({
                'head': self._getBlockId(connection.head),
                'tail': self._getBlockId(connection.tail),
                'type': self._getEnumId('ConnectionType', connection.type)
            })
            self.execute(query)
        pass
    def _getBlockId(self, block) -> int:
        table = Block.getTable()
        query = table.getSelectSQL(fields=['id'], conditions={
            'name': self._handldApostropheString(block.name),
            'content': self._handldApostropheString(block.content),
            'prediction': self._handldApostropheString(block.prediction),
            'type': self._getEnumId('BlockType', block.type)
        })
        res = self.execute(query)
        return res[0][0]
    def _getEnumId(self, enum, enumName: str) -> int:
        # base on enumname to get blocktype or connectiontype id
        if enum in globals():
            enumClass = globals()[enum]
            table = enumClass.getTable()
            query = table.getSelectSQL(fields=['id'], conditions={
                'name': enumName
            })
            res = self.execute(query)
            return res[0][0]
        return 0
    def _getEnumName(self, enum, enumId: int) -> str:
        if enum in globals():
            enumClass = globals()[enum]
            table = enumClass.getTable()
            query = table.getSelectSQL(fields=['name'], conditions={
                'id': enumId
            })
            res = self.execute(query)
            return res[0][0]
        return ''
    def _handldApostropheString(self, string: str) -> str:
        return string.replace("'", "''")
        pass
```

Listing A.21: DBMS class.