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. Al-

though 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

LLMs approach compared to fomulated 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.

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.

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.

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.

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.

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	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

Overall, this system have two separate implementation: User Interface (Frontend) and Application service (Backend), connected through API Gateway. In this thesis, we will focus on how the service and each component inside is design.

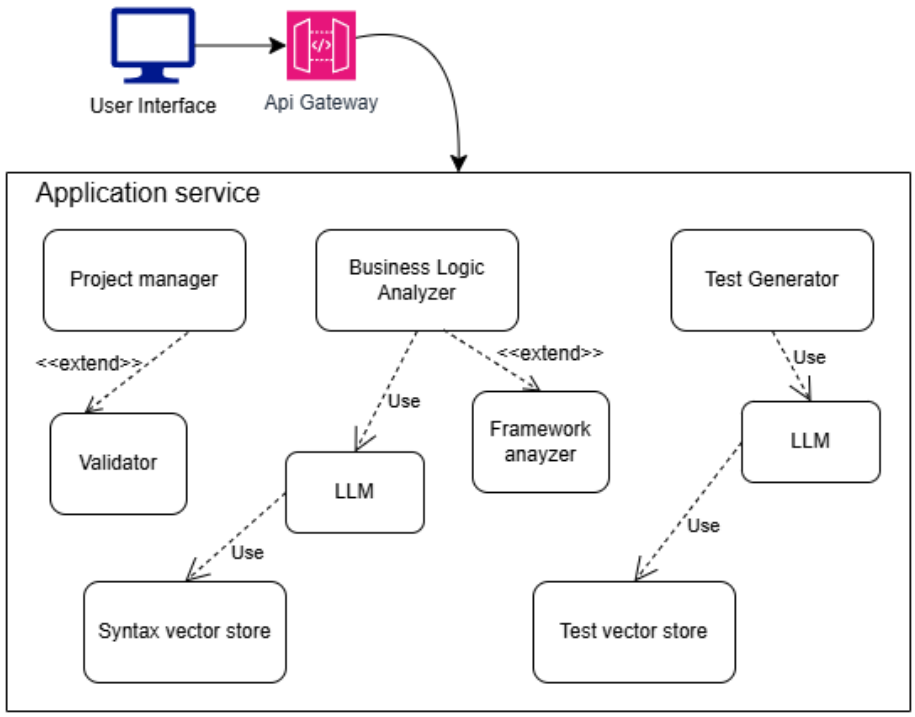

Figure 3.1: Test Genie's overall component design

Project manager. This component is abstracted by framework, handle anything related to the project's files. This component also interact with Git to clone the required project and responsible to use the SDK to validate existing tests.

BLA module. The BLA module plays a crucial role in analyzing the project's source code. It interacts with both project files and the database to break down the source code into components and identify relationships between classes and functions. These relationships are stored in a SQL database and visualized as component dependency diagrams to provide users with a clear representation of the project's structure. Analyzing strategy in this module will also be abstracted by framework.

Test Generator. This component communicates with Large Language Models (LLMs) to generate test files based on the analyzed business logic. It uses prompts and additional embedded documentation to produce runnable and framework-compliant test files that meet the project's requirements.

Block of component ERD. To analyze user's project easier, BLA module will firstly split the source files into components. These components will be connected together through different type of connection.

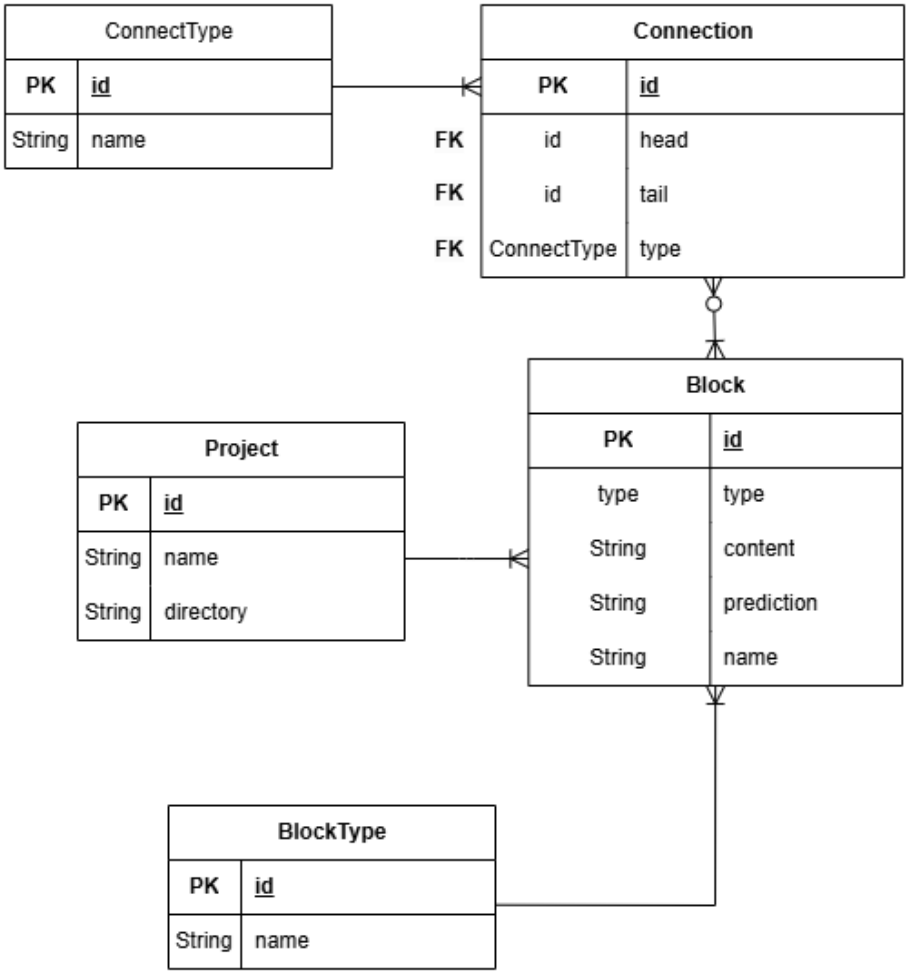

Figure 3.2: Block Relational Database Design

At the core of the diagram is the *Block entity*, which represents distinct units of the source code identified during the project analysis. Each components stores attributes such as its type, content, prediction, and a reference to the project it belongs to. The block entity will not need to store the id of its project because it is stored as system files in the server and Backend can access it directly. This kind of design will reduce the size of the database and optimize the performance of the system.

The *Connection entity* defines the relationships between blocks. It uses references to two distinct blocks: head and tail, forming a directional link between them. These connections are categorized by the ConnectType entity, which stores different types of connections that can exist between blocks, such as data flow, control flow, or dependency relationships. This architecture facilitates a comprehensive understanding of the project's code structure by mapping both the functional and logical connections between different blocks.

Furthermore, the *BlockType entity* is used to define the classification of blocks, storing various block types such as files, classes or functions. This separation of component types allows for better categorization and analysis. The Project entity ensures that each component is tied to a specific source code directory, while ConnectType maintains clarity by classifying relationships between them. This ERD structure enables the BLA module to effectively visualize and analyze complex relationships within user projects, making it easier to identify patterns and dependencies for test generation.

Just like *BlockType entity*, the *ConnectType entity* is used to define the classification of connections, storing various connection types such as Call, Contain, Use relationships, etc. This separation of connection types allows for better categorization and analysis. The Project entity ensures that each component is tied to a specific source code directory, while ConnectType maintains clarity by classifying relationships between them. This ERD structure enables the BLA module to effectively visualize and analyze complex relationships within user projects, making it easier to identify patterns and dependencies for test generation.

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.

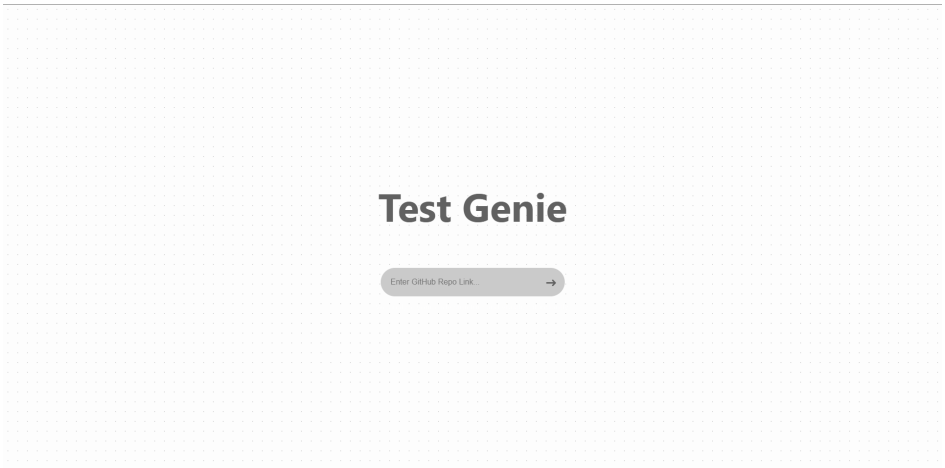

Figure 4.1: Homepage of Test Genie system.

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.

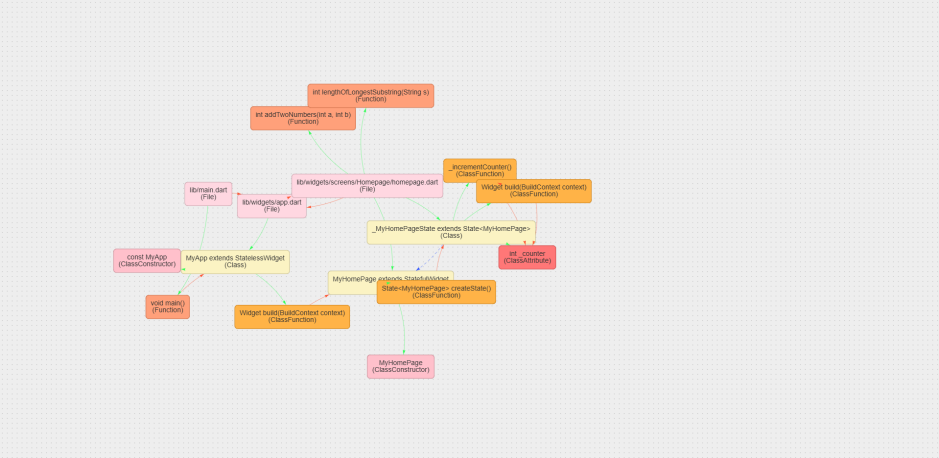

Figure 4.2: Initial load of the dependency diagram.

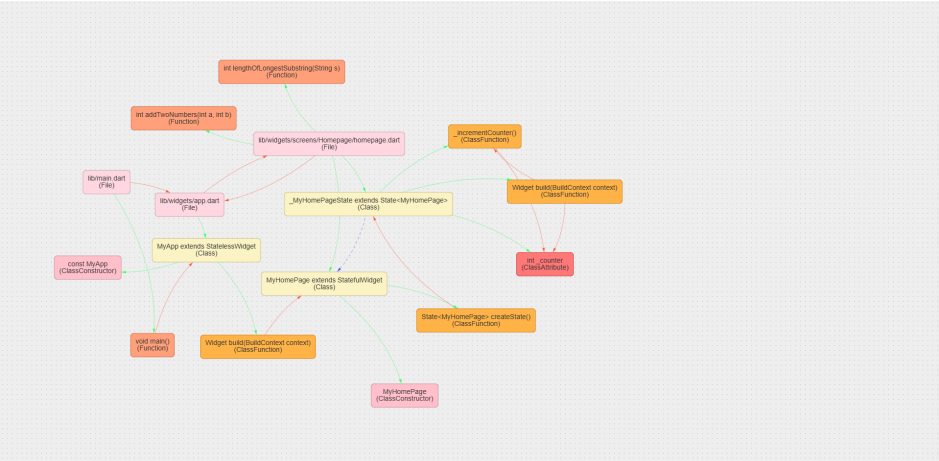

Figure 4.3: Diagram blocks can be dragged to rearrange their positions.

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.

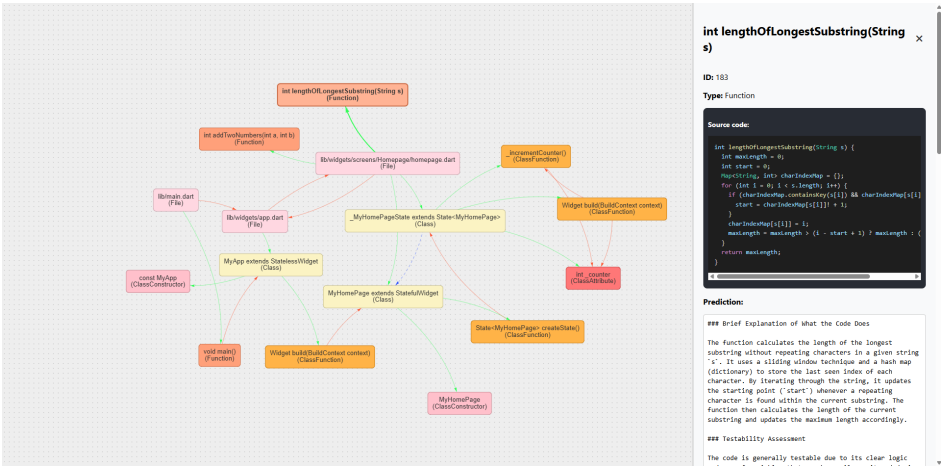

Figure 4.4: Block Detail View showcasing the block's content and prediction.

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.

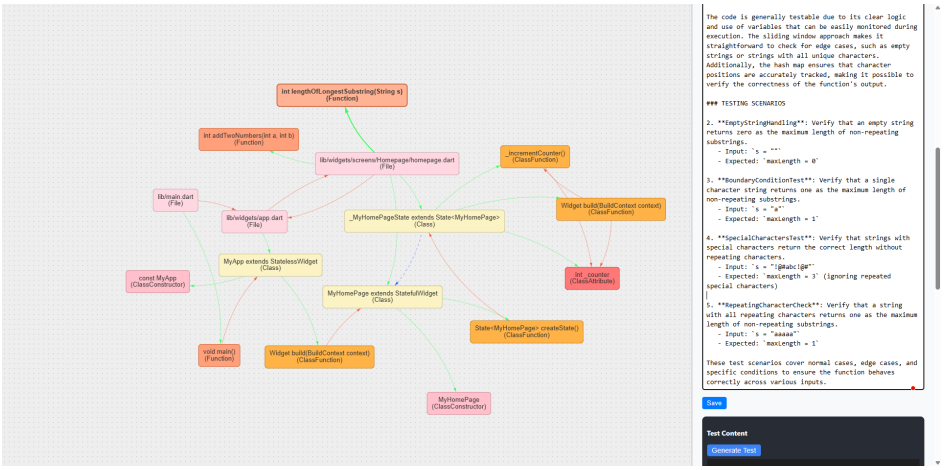

Figure 4.5: Prediction adjustment interface for refining AI-generated predictions.

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.

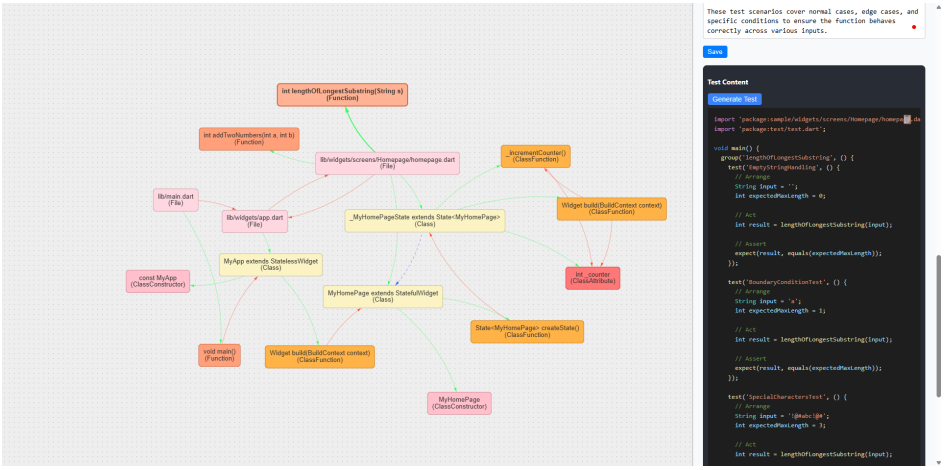

Figure 4.6: Generated test cases for a specific block.

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

5.1 Performance Analysis

5.1.1 AI generation time

5.1.2 BLA Algorithm complexity

Time Complexity

Space Complexity

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.

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
```

```
46          try:
47              self._runFlutterCLI('--version', isRaiseException=True)
48          except subprocess.CalledProcessError as e:
49              raise Exception(f'Error checking flutter sdk: {e}')
50
51          # print(result)
52
53      def _getYamlName(self) -> str:
54
55          yamlContent = self.getFileContent('pubspec.yaml')
56          # print(yamlContent)
57          # first line should define the name of the project: "name:
    ....."
58          return yamlContent.split('\n')[0].split('name: ')[1].strip()
59
60
61      # function for testing only. Do not use in production
62      def _createSampleProject(self, prjName) -> str:
63          try:
64              # cannot use _runFlutterCLI because no project directory
    yet
65              # result = self._runFlutterCLI(['create', prjName],
    isRaiseException=True)
66              result = subprocess.check_output([os.path.join(sdkDir, '
    bin', 'flutter'), 'create', prjName],cwd=projectDir,
    universal_newlines=True, encoding='utf-8',  shell=True)
67
68          except subprocess.CalledProcessError as e:
69              raise Exception(f'Error creating flutter project: {e}')
70          return result
71
72      def _flutterPubGet(self) -> None:
73          # prjDir = os.path.join(projectDir, self.getName())
74          # flutterBatDir = os.path.join(sdkDir, 'bin', 'flutter.bat')
75
76          try:
77              # result = subprocess.check_output([flutterBatDir, 'pub',
    'get'], cwd=prjDir, universal_newlines=True)
78              self._runFlutterCLI(['pub', 'get', '--no-example'],
    isRaiseException=True)
79          except subprocess.CalledProcessError as e:
80              raise Exception(f'Error running flutter pub get: {e}')
81
82          # print(result)
83
84      def _addTestDependency(self) -> None:
85          # run pub add test
86          try:
87              self._runFlutterCLI(['pub', 'add', 'test'],
    isRaiseException=True)
88          except subprocess.CalledProcessError as e:
89              raise Exception(f'Error adding test dependency: {e}')
90          # print(result)
91
92      def create_test(self, filename, content, isOverWrite = False) ->
    None:
93          # create test file in the test directory
94          # check if test directory exists
95          testDir = os.path.join(projectDir, self.getName(), 'test')
```

```
96          if not os.path.exists(testDir):
97              os.makedirs(testDir)
98          # check if file exists
99          fileDir = os.path.join(testDir, filename)
100         if os.path.exists(fileDir) and not isOverWrite:
101             raise Exception(f'File {fileDir} already exists')
102         # create file
103         with open(fileDir, 'w') as f:
104             f.write(content)
105
106     def get_test_content(self, filename) -> str:
107         # use getFileContent to get the content of the test file
108         testDir = os.path.join(projectDir, self.getName(), 'test')
109         fileDir = os.path.join(testDir, filename)
110         if not os.path.exists(fileDir):
111             raise Exception(f'File {fileDir} does not exist')
112         return self.getFileContent(fileDir)
113
114     # return tuple (result, error)
115     def run_test(self, filename) -> tuple:
116         fileDir = os.path.join('test', filename)
117         try:
118             result = self._runFlutterCLI(['test', fileDir])
119         except subprocess.CalledProcessError as e:
120             raise Exception(f'Error running flutter test: {e}')
121         return result
122         pass
123
124     def validate(self) -> str:
125         # run all tests in the test directory
126         testDir = os.path.join(projectDir, self.getName(), 'test')
127         for file in os.listdir(testDir):
128             if file.endswith('.dart'):
129                 result, err = self.run_test(file)
130                 if err:
131                     return err
132
133         return ''
134
135     def getListSourceFiles(self) -> list[str]:
136             """_summary_
137
138             Returns:
139                 list[str]: list of source files in the project
    relative to project directory
140             """
141             prjDir = os.path.join(projectDir, self.getName())
142             libDir = os.path.join(prjDir, 'lib')
143             sourceFiles = []
144
145             # find main.dart first
146             if os.path.exists(os.path.join(libDir, 'main.dart')):
147                 sourceFiles.append(os.path.relpath(os.path.join(libDir
    , 'main.dart'), prjDir))
148
149             for root, dirs, files in os.walk(libDir):
150                 for file in files:
151                     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:
```

```
15              # if \n, reset isCommentSingleLine
16              if content[i] == '\n':
17                  isCommentSingleLine = False
18              if content[i] == '/' and content[i+1] == '*':
19                  isCommentMultiLine = True
20              if content[i] == '/' and content[i+1] == '/':
21                  isCommentSingleLine = True
22              if not isCommentSingleLine and not isCommentMultiLine:
23                  res += content[i]
24              if content[i] == '*' and content[i+1] == '/':
25                  isCommentMultiLine = False
26                  i+=1
27              i+=1
28
29          # delete all empty lines
30          res = '\n'.join([line for line in res.split('\n') if line.
    strip() != ''])
31
32          return res
33
34      def setPrediction(self, prediction: str) -> None:
35          self.prediction = prediction
36
37      def getPrediction(self) -> str:
38          return self.prediction
```

Listing A.4: `Block` class.

```
1       class BlockType:
2           FILE = 'File'
3           CLASS = 'Class'
4           ABSTRACT_CLASS = 'AbstractClass'
5           ENUM = 'Enum'
6           GLOBAL_VAR = 'GlobalVar'
7           FUNCTION = 'Function'
8           CLASS_CONSTRUCTOR = 'ClassConstructor'
9           CLASS_FUNCTION = 'ClassFunction'
10          CLASS_ATTRIBUTE = 'ClassAttribute'
```

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

```
1       class Connection:
2       def __init__(self, head: Block, tail: Block, type: str):
3           self.head = head
4           self.tail = tail
5           self.type = type
```

Listing A.6: `Connection` class.

```
1       class ConnectionType:
2           EXTEND = 'Extend'
3           IMPLEMENT = 'Implement'
4           CONTAIN = 'Contain'
5           EXTEND = 'Extend'
6           USE = 'Use'
7           CALL = 'Call'
8           IMPORT = 'Import'
```

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

```
    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.

```
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))
```

```
31                      else: diagram.connections.append(Connection(block,
     [b for b in diagram.blocks if b.name == fileDir][0],
    ConnectionType.IMPORT))
32              else:
33                  # import as relative path
34                  currentDir = block.name #ex: lib/main.dart
35                  currentDir = currentDir.split('/')
36                  currentDir.pop()
37                  currentDir = '/'.join(currentDir)
38                  combineDir = os.path.normpath(os.path.join(currentDir,
     directory))
39                  # print(combineDir)
40                  fileContent = diagram.project.getFileContent(
    combineDir)
41                  if combineDir not in [block.name for block in diagram.
    blocks]:
42                      # turn \ into /
43                      combineDir = combineDir.replace('\\','/')
44                      blocks.append(Block(combineDir, fileContent,
    BlockType.FILE))
45                  else: diagram.connections.append(Connection(block, [b
    for b in diagram.blocks if b.name == combineDir][0],
    ConnectionType.IMPORT))
46
47          for b in blocks:
48              # print(b)
49              diagram.blocks.append(b)
50              diagram.connections.append(Connection(block, b,
    ConnectionType.IMPORT))
51              ImportAnalyzer(diagram, b)
```

Listing A.9: `ImportAnalyzer` function.

```
1  def ContainAnalyzer(diagram, block, visited = []):
2      visited.append(block)
3      currType = block.type
4
5      # print("Current content: ", currContent)
6      # print("Current type: ", currType)
7
8      # keep analyze if type is file or class or abstract class
9      if (currType == BlockType.FILE
10         or currType == BlockType.CLASS
11         or currType == BlockType.ABSTRACT_CLASS
12         ):
13         content = block.getContentNoComment()
14         # print(content)
15         lines = content.split('\n')
16         # if type is file, analyze classes and functions (standalone
    functions)
17         # if type is class, analyze functions
18         blocks = []
19         # File analyzing
20         if (currType == BlockType.FILE):
21             # two cases: class and abstract class
22             if 'class ' in content:
23                 # this file have class(es)
24                 isClassContent = False
25                 openedBracket = 0
26                 className = ''
```

```
27                  classContent = []
28                  # class or final class
29                  for line in lines:
30                      if line.strip().startswith('class ') or line.strip
    ().startswith('final class '):
31                          # first line of class
32                          # NOTE: there is no class inside class
33                          # get class name
34                          className = line.split('class ')[1].split('{')
    [0].strip()
35                          # print(className)
36                          isClassContent = True
37                          classContent.append(line)
38                      elif '}' in line and isClassContent:
39                          # two cases: class end or function end
40                          classContent.append(line)
41                          if '{' in line:
42                              continue
43                          if openedBracket > 0:
44                              openedBracket = openedBracket - 1
45                          else:
46                              # class end
47                              isClassContent = False
48                              classContent = '\n'.join(classContent)
49                              blocks.append(Block(className,
    classContent, BlockType.CLASS))
50                              classContent = []
51                      elif '{' in line and isClassContent:
52                          openedBracket = openedBracket + 1
53                          classContent.append(line)
54                      elif isClassContent:
55                          classContent.append(line)
56
57                  # abstract class
58                  if 'abstract class ' in content or 'abstract final
    class ' in content:
59                      isAbstractClassContent = False
60                      openedBracket = 0
61                      className = ''
62                      classContent = []
63                      for line in lines:
64                          if line.strip().startswith('abstract class ')
    or line.strip().startswith('abstract final class '):
65                              # first line of class
66                              # NOTE: there is no class inside class
67                              # get class name
68                              if line.strip().startswith('abstract class
     '):
69
70                                  className = line.split('abstract class
     ')[1].split('{')[0].strip()
71                              # Check if it's an abstract final class
72                              if line.strip().startswith('abstract final
     class '):
73                                  className = line.split('abstract final
     class ')[1].split('{')[0].strip()
74                              # print(className)
75                              isAbstractClassContent = True
76                              classContent.append(line)
```

```
77                          elif '}' in line and isAbstractClassContent:
78                              # two cases: class end or function end
79                              classContent.append(line)
80                              if '{' in line:
81                                  continue
82                              if openedBracket > 0:
83                                  openedBracket = openedBracket - 1
84                              else:
85                                  # class end
86                                  isAbstractClassContent = False
87                                  classContent = '\n'.join(classContent)
88                                  blocks.append(Block(className,
    classContent, BlockType.ABSTRACT_CLASS))
89                                  classContent = []
90                          elif '{' in line and isAbstractClassContent:
91                              openedBracket = openedBracket + 1
92                              classContent.append(line)
93                          elif isAbstractClassContent:
94                              classContent.append(line)
95
96              # enum
97              if 'enum ' in content:
98                  # no {} in enum
99                  # maybe () in enum
100                 # once } is found, enum end
101                 isEnumContent = False
102                 enumName = ''
103                 enumContent = []
104                 for line in lines:
105                     if line.strip().startswith('enum '):
106                         # first line of enum
107                         enumName = line.split('enum ')[1].split('{')
    [0].strip()
108                         isEnumContent = True
109                         enumContent.append(line)
110                     elif '}' in line and isEnumContent:
111                         enumContent.append(line)
112                         isEnumContent = False
113                         enumContent = '\n'.join(enumContent)
114                         blocks.append(Block(enumName, enumContent,
    BlockType.ENUM))
115                         # print(enumContent)
116                     elif isEnumContent:
117                         enumContent.append(line)
118
119             # function
120             # standalone function / GlobalVar only!
121             # strat: get rid of all analyzed class and enum first
122             leftoverContent = content
123             for b in blocks:
124                 leftoverContent = leftoverContent.replace(b.content, '
    ')
125             # get rid of import line
126             leftoverContent = '\n'.join([line for line in
    leftoverContent.split('\n') if not line.strip().startswith('import
    ')])
127             # remove empty lines
128             leftoverContent = '\n'.join([line for line in
    leftoverContent.split('\n') if line.strip() != ''])
```

```
129             # print("========Leftover content========")
130             # print(block.name)
131             # print(leftoverContent)
132 
133             # two case of function: difined return type or not (
    dynamic)
134             # variable must have a type
135             # print("========Function and GlobalVar========")
136             funcAndVarBlocks = extract_functions_and_globals(
    leftoverContent)
137             blocks.extend(funcAndVarBlocks)
138 
139         # Class analyzing
140         if (currType in (BlockType.CLASS)):
141             # two cases: class function and class attribute
142             content = block.getContentNoComment() # should be no
    difference between content and contentNoComment
143             # print(content)
144             classContentBlock = extract_class_content(content)
145             blocks.extend(classContentBlock)
146 
147         # blocks recursive
148         for b in blocks:
149             # print(b)
150             # print(b.content)
151             diagram.blocks.append(b)
152             diagram.connections.append(Connection(block, b,
    ConnectionType.CONTAIN))
153             ContainAnalyzer(diagram, b, visited=visited)
154 
155     # find connection connected to this block and not visited
156     connectedBlocks = [c.tail for c in diagram.connections if c.head
    == block and c.tail not in visited]
157     for b in connectedBlocks:
158         ContainAnalyzer(diagram, b, visited=visited)
```

Listing A.10: `ContainAnalyzer` function.

```
1     def CallAnalyzer(diagram, block, visited = []):
2     if block in visited:
3         return
4 
5     visited.append(block)
6     currType = block.type
7 
8     # NOTE strat: 2-layer recursive
9     if currType in (BlockType.FILE):
10        # find connected file (imported file)
11        connectedFiles = [conn.tail for conn in diagram.connections if
     conn.head == block and conn.type == ConnectionType.IMPORT]
12 
13        for file in connectedFiles:
14            # print("Imported file:")
15            # print(file)
16            # find all class/function/variable in file. Avoid
    BlockType.FILE
17            connectedBlocks = [conn.tail for conn in diagram.
    connections if conn.head == file and conn.type == ConnectionType.
    CONTAIN]
```

```
18              currentBlocks = [conn.tail for conn in diagram.connections
     if conn.head == block and conn.type == ConnectionType.CONTAIN]
19
20
21              _CallAnalyzer(diagram, currentBlocks, connectedBlocks,
    visited)
22              # import based recursive call
23              CallAnalyzer(diagram, file, visited)
24
25 def _CallAnalyzer(diagram, thisFile, callables, visited=[]):
26      # thisFile: blocks of contains in current file
27      # callables: blocks of contains in imported file
28      callables.extend(thisFile)
29
30      # printStuff(thisFile, callables)
31
32      for block in thisFile:
33          if block in visited:
34              continue
35
36          if block.type in (BlockType.ABSTRACT_CLASS, BlockType.CLASS):
37              # extend connection analyze
38              name = block.name
39              # print(name)
40              # first word is class name
41              classname = name.split()[0]
42              otherInfo = name[len(classname):]
43              for connBlock in callables:
44                  if connBlock.type in (BlockType.ABSTRACT_CLASS,
    BlockType.CLASS):
45                      className = connBlock.name.split()[0]
46                      # if className found in otherInfo, create a
    connection ConnectionType.EXTEND
47                      if className in otherInfo:
48                          diagram.connections.append(Connection(block,
    connBlock, ConnectionType.EXTEND))
49                          # print(f"Extend connection: {block} --> {
    connBlock}")
50
51              # split class, abstract class
52              innerBlocks = [conn.tail for conn in diagram.connections
    if conn.head == block and conn.type == ConnectionType.CONTAIN]
53              # magic recursive calls at 4 a.m
54              visited.append(block)
55              _CallAnalyzer(diagram, innerBlocks, callables, visited)
56
57
58              continue
59          else:
60              visited.append(block)
61              # analyze calls in block
62              # If called, create a connection ConnectionType.CALL
63
64              fullcontent = block.getContentNoComment()
65              # print("====================================")
66              # print(block)
67              # print(fullcontent)
68
69              content = ''
```

```
70              # Extract content only, exclude function name, params
71              if block.type in (BlockType.FUNCTION, BlockType.
    CLASS_FUNCTION):
72                  if '=>' in fullcontent:
73                      # take content from => to ;
74                      # add a ; to the end of content
75                      fullcontent = fullcontent + ';'
76                      content = fullcontent[fullcontent.index('=>')+2:]
77                      content = content[:content.index(';') + 1]
78
79                  else:
80                      roundbracketOpened = 0
81                      initialRoundBracket = False
82                      curlybracketOpened = 0
83                      isContent = False
84                      for char in fullcontent:
85                          # params section
86                          if char == '(' and not isContent and not
    initialRoundBracket:
87                              initialRoundBracket = True
88                              roundbracketOpened += 1
89                          if char == ')' and not isContent:
90                              roundbracketOpened -= 1
91                              if roundbracketOpened == 0 and
    initialRoundBracket:
92                                  initialRoundBracket = False
93                          if char == '{':
94                              if not isContent and not
    initialRoundBracket:
95                                  isContent = True
96                              curlybracketOpened += 1
97                          if char == '}' and isContent:
98                              curlybracketOpened -= 1
99                              if curlybracketOpened == 0:
100                                 isContent = False
101                                 content += char
102                                 break
103                         if isContent:
104                             content += char
105             if block.type in (BlockType.CLASS_ATTRIBUTE):
106                 # extract content from = to ;
107                 # add a ; to the end of content
108                 fullcontent = fullcontent + ';'
109                 content = fullcontent[fullcontent.index('=')+1:]
110                 content = content[:content.index(';') + 1]
111             # print("====================Block====================")
112             # print("Block name: ", block.name)
113             # print("====================Extracted content
    ====================")
114             # print(content)
115             # print
    ("============================================================")
116             # printStuff(thisFile, callables)
117             callablesName = getCallablesName(callables)
118             for name, connBlock in callablesName:
119                 # print(f"Name: {name}, Block name: {connBlock.name}")
120                 # find name in content
121                 # name found can be next to any non-word character or
    start of line and end of line
```

```
                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.

```
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, "
```

```
90              "formulate a statement that can be used to query the model
     for useful reference."
91              "Do NOT include the user request in the query."
92              # "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."
93          )
94          contextualize_q_prompt = ChatPromptTemplate.from_messages(
95              [
96                  ("system", contextualize_q_system_prompt),
97                  MessagesPlaceholder("chat_history"),
98                  ("human", "{input}"),
99              ]
100         )
101         # Create a history-aware retriever
102         # This uses the LLM to help reformulate the question based on
    chat history
103         history_aware_retrievers = []
104
105         for retriever in self.retrievers:
106             history_aware_retrievers.append(
107                 create_history_aware_retriever(
108                     self.model, retriever, contextualize_q_prompt
109                 )
110             )
111
112         bla_system_prompt = (
113             "You are an AI assistant that analyzes Flutter/Dart source
     code to identify its business logic for test generation.\n"
114             "You can provide helpful answers using available tools.\n"
115             "For the given code snippet:\n\n"
116             "1. FUNCTION ANALYSIS:\n"
117             "   - What is the purpose of this function/class?\n"
118             "   - What are the inputs (parameters) and their types?\n"
119             "   - What is the expected output (return value) and its
    type?\n"
120             "   - What algorithm or logic does it implement?\n\n"
121             "2. TESTABILITY ASSESSMENT:\n"
122             "   - Can this code be tested? If yes, what type of test
    is appropriate (unit/widget/integration)?\n"
123             "   - Are there any dependencies that might complicate
    testing?\n\n"
124             "3. TESTING SCENARIOS:\n"
125             "   ALWAYS include at least 3-5 specific test scenarios
    using EXACTLY this format:\n\n"
126             "   TESTING SCENARIOS:\n"
127             "   1. [Scenario Name]: Verify that [functionality]. Input
    : [specific input values]. Expected: [specific output/behavior].\n
    "
128             "   2. [Scenario Name]: Verify that [functionality]. Input
    : [specific input values]. Expected: [specific output/behavior].\n
    "
129             "   3. [Scenario Name]: Verify that [functionality]. Input
    : [specific input values]. Expected: [specific output/behavior].\n
    \n"
130             "Keep your analysis concise but precise. DO NOT include
    the source code in your answer.\n"
131             "The TESTING SCENARIOS section MUST follow the exact
    format shown above, with specific input values and expected
```

```
    outputs.\n"
132             "If the code's purpose is unclear, make your best
    inference based on the implementation details.\n"
133             "{context}"
134         )
135
136         bla_prompt = ChatPromptTemplate.from_messages(
137             [
138                 ("system", bla_system_prompt),
139                 MessagesPlaceholder("chat_history"),
140                 ("human", "{input}"),
141             ]
142         )
143         bla_chain = create_stuff_documents_chain(self.model,
    bla_prompt)
144
145         rag_chains = []
146         for retriever in history_aware_retrievers:
147             rag_chains.append(create_retrieval_chain(retriever,
    bla_chain))
148
149         react_docstore_prompt = hub.pull("hwchase17/react")
150
151         tools = []
152
153         store_names = []
154         for store_name, doc_name in self.store_names.items():
155             store_names.append(store_name)
156
157         for i in range(len(store_names)):
158             # print(f"{store_names[i]}")
159             tools.append(
160                 Tool(
161                     name=f"Get code explaination from {store_names[i]}
    ",
162                     func=lambda input, **kwargs: rag_chains[i].invoke(
163                         {
164                             "input": input,
165                             "chat_history": kwargs.get("chat_history",
     [])
166                         }
167                     ),
168                     description=f"Retrieve documents from the vector
    store {store_names[i]}",
169                 )
170             )
171         agent = create_react_agent(
172             llm=self.model,
173             tools=tools,
174             prompt=react_docstore_prompt,
175         )
176
177         self.agent_executor = AgentExecutor.from_agent_and_tools(
178             agent=agent,
179             tools=tools,
180             handle_parsing_errors=True,
181             verbose=True,
182         )
183         pass
```

```
184     # Function to create and persist vector store
185     def _create_vector_store(self, docs, store_name, is_overwrite=
    False) -> None:
186         persistent_directory = os.path.join(db_dir, store_name)
187         # delete the directory if it exists and needed
188         if is_overwrite and os.path.exists(persistent_directory):
189             shutil.rmtree(persistent_directory) # remove the directory
190 
191         if not os.path.exists(persistent_directory):
192             print(f"\n--- Creating vector store {store_name} ---")
193             db = Chroma.from_documents(
194                 docs, self.embeddings, persist_directory=
    persistent_directory
195             )
196             print(f"--- Finished creating vector store {store_name}
    ---")
197         else:
198             print(
199                 f"Vector store {store_name} already exists. No need to
     initialize.")
200     def _load_document(self, doc_name):
201         file_path = os.path.join(docs_dir, doc_name)
202         if not os.path.exists(file_path):
203             raise FileNotFoundError(
204                 f"_load_document: The file {file_path} does not exist.
     Please check the path."
205             )
206         file_extension = os.path.splitext(file_path)[1]
207         # check if the file extension is supported
208         if file_extension not in file_loader_map:
209             raise Exception(f"_load_document: Unsupported file
    extension: {file_extension} for file: {file_path}")
210         loader = PyPDFLoader(file_path=file_path)
211         return loader.load()
212 
213     def _split_document(self, documents, chunk_size=1000,
    chunk_overlap=100):
214         text_splitter = SentenceTransformersTokenTextSplitter(
215             chunk_size=chunk_size, chunk_overlap=chunk_overlap
216         )
217         return text_splitter.split_documents(documents)
218 
219     def _check_if_vector_store_exists(self, store_name) -> bool:
220         persistent_directory = os.path.join(db_dir, store_name)
221         return os.path.exists(persistent_directory)
```

Listing A.12: `AI_Agent` class.

```
1     OPENAI_API_KEY=sk-this-key-is-just-a-placeholder
2     LANGCHAIN_API_KEY=sk-this-key-is-just-a-placeholder
3     LANGCHAIN_PROJECT=TestGenie
4 
5     BASE_URL=
6     BLA_LLM_MODEL=
7     TG_LLM_MODEL=
8 
9     EMBED_MODEL=
```

Listing A.13: `Sample .env` file.

```
    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.
```

```
57
58          Args:
59              package_name: Name of the package (for import statements)
60              code_location: Location of the code file to test (path
    within the package)
61              function_name_and_arguments: Function signature with
    arguments
62              prediction: Description of what the function does
63
64          Returns:
65              Generated test case as a string (clean Dart source code
    only)
66          """
67          try:
68          # Extract the structured sections from prediction if needed
69          # Check if prediction contains the expected structure
70              if "TESTING SCENARIOS:" not in prediction and "Brief" not
    in prediction:
71                  # If prediction isn't properly structured, try to
    structure it
72                  structured_prompt = (
73                      "Structure this analysis into the following format
    :\n"
74                      "1. Brief explanation of what the code does\n"
75                      "2. Testability assessment\n"
76                      "3. TESTING SCENARIOS in this exact format:\n\n"
77                      "TESTING SCENARIOS:\n"
78                      "1. [Descriptive Test Name]: Verify that [
    functionality]. Input: [specific input values]. Expected: [
    specific output/behavior].\n"
79                      "2. [Descriptive Test Name]: Verify that [
    functionality]. Input: [specific input values]. Expected: [
    specific output/behavior].\n"
80                      "3. [Descriptive Test Name]: Verify that [
    functionality]. Input: [specific input values]. Expected: [
    specific output/behavior].\n\n"
81                      "Analysis to structure: " + prediction
82                  )
83
84                  structured_response = self.model.invoke(
    structured_prompt)
85                  prediction = structured_response.content
86
87              # Generate the test case with the structured prediction
88              raw_output = self._generate_clean_test(package_name,
    code_location, function_name_and_arguments, prediction)
89
90              # Clean up any markdown formatting that might be present
91              cleaned_output = self._clean_code_output(raw_output)
92
93              return cleaned_output
94          except Exception as e:
95              print(f"Error generating test case: {str(e)}")
96              return f"// Error generating test case: {str(e)}"
97
98      def fix_generated_code(
99          self,
100         error_message: str,
101         current_test_code: str,
```

```
102         prediction: str,
103     ) -> str:
104         """
105         Fix issues in generated test code based on error messages from the Dart SDK.
106         Enhanced with online search and error pattern learning capabilities.
107 
108         Args:
109             error_message: The error message from the Dart SDK
110             current_test_code: The current test code that has issues
111             prediction: The original prediction about what the function does
112 
113         Returns:
114             Fixed test code that addresses the errors
115         """
116         try:
117             # Create a unique identifier for this error+code combination to track fix attempts
118             error_hash = self._generate_error_hash(error_message, current_test_code)
119 
120             # Check if we've seen and fixed this exact error before
121             if error_hash in self.error_fix_cache:
122                 print(f"Using cached fix for error: {error_hash[:10]}...")
123                 return self.error_fix_cache[error_hash]
124 
125             # Track fix attempts to avoid infinite loops
126             if error_hash not in self.attempted_fixes_for_error:
127                 self.attempted_fixes_for_error[error_hash] = 0
128 
129             self.attempted_fixes_for_error[error_hash] += 1
130 
131             # If we've tried too many times, use different strategies or bail out
132             if self.attempted_fixes_for_error[error_hash] > self.max_error_fix_attempts:
133                 print(f"Maximum fix attempts reached for error {error_hash[:10]}. Applying emergency fix...")
134                 # Apply emergency fix that attempts to produce at least a basic test case
135                 emergency_fixed = self._emergency_fix(current_test_code, error_message)
136                 self.error_fix_cache[error_hash] = emergency_fixed
137                 return emergency_fixed
138 
139             # Extract unique errors from the potentially repetitive error message
140             unique_errors = self._extract_unique_errors(error_message)
141 
142             # 1. First try our standard approach
143             if self.attempted_fixes_for_error[error_hash] == 1:
144                 fixed_code = self._standard_ai_fix(unique_errors, current_test_code)
145 
146             # 2. If that didn't work, search online for solutions
147             elif self.attempted_fixes_for_error[error_hash] == 2:
```

```
148                 # Search for online solutions for this error
149                 online_solutions = self._search_for_error_solutions(
    unique_errors)
150 
151                 # Use online solutions to enhance fix prompt
152                 fixed_code = self._ai_fix_with_online_knowledge(
    unique_errors, current_test_code, online_solutions)
153 
154             # 3. Final attempt with different approach
155             else:
156                 fixed_code = self._comprehensive_repair_attempt(
    error_message, current_test_code, prediction)
157 
158             # Apply additional specific rule-based fixes
159             fixed_code = self._apply_specific_fixes(fixed_code,
    unique_errors)
160 
161             # Cache the successful fix for this error
162             self.error_fix_cache[error_hash] = fixed_code
163 
164             return fixed_code
165 
166         except Exception as e:
167             print(f"Error fixing test code: {str(e)}")
168             # Try a simpler approach with manual fixes for common
    errors
169             try:
170                 manually_fixed = self._manual_fix_common_errors(
    current_test_code, error_message)
171                 return manually_fixed
172             except:
173                 # If all else fails, return the original with error
    comments
174                 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.

```
1     frameworkMap = {
2         'flutter': Flutter
3     }
4 
5     def getDBMS(git_url) -> DBMS:
6         project = Project(git_url)
7         framework = project.recognizeProjectFramework()
8 
9         if framework in frameworkMap:
10            project = frameworkMap[framework](git_url)
11 
12        dbms = DBMS(project)
13 
14        return dbms
15 
16    app = Flask(__name__)
17    CORS(app)
18 
19    # Post git project url
20    @app.route('/createProject', methods=['POST'])
21    def createProject():
```

```
22          if not request.json or not 'git_url' in request.json:
23              return jsonify({'message': 'Invalid request'})
24          git_url = request.json['git_url']
25          project = Project(git_url)
26          # print(project)
27          return jsonify({'message': f'{project}'})
28
29      @app.route('/getDiagram', methods=['POST'])
30      def getDiagram():
31          # print(request.json)
32          if not request.json or not 'git_url' in request.json:
33              return jsonify({'message': 'Invalid request'})
34          git_url = request.json['git_url']
35
36          dbms = getDBMS(git_url)
37
38          diagram = dbms.getJsonDiagram()
39          return jsonify(diagram)
40
41      @app.route('/getDiagram', methods=['OPTIONS'])
42      def getDiagramOptions():
43          print(request.json)
44          print("wrong method")
45          return jsonify({'message': 'Options request'})
46
47      @app.route('/getBlockContent', methods=['POST'])
48      def getBlockContent():
49          if not request.json or not 'git_url' in request.json or not '
        block_id' in request.json:
50              return jsonify({'message': 'Invalid request'})
51          git_url = request.json['git_url']
52          blockId = request.json['block_id']
53
54          dbms = getDBMS(git_url)
55          blockContent = dbms.getBlockContent(blockId)
56          return jsonify(blockContent)
57
58      @app.route('/getBlockPrediction', methods=['POST'])
59      def getBlockPrediction():
60          if not request.json or not 'git_url' in request.json or not '
        block_id' in request.json:
61              return jsonify({'message': 'Invalid request'})
62          git_url = request.json['git_url']
63          blockId = request.json['block_id']
64
65          dbms = getDBMS(git_url)
66          blockPrediction = dbms.getBlockPrediction(blockId)
67          return jsonify(blockPrediction)
68
69      @app.route('/getBlockDetail', methods=['POST'])
70      def getBlockDetail():
71          if not request.json or not 'git_url' in request.json or not '
        block_id' in request.json:
72              return jsonify({'message': 'Invalid request'})
73          git_url = request.json['git_url']
74          blockId = request.json['block_id']
75
76          dbms = getDBMS(git_url)
77          # {
```

```
78              # 'content': blockContent,
79                # 'prediction': blockPrediction,
80          # }
81          content = dbms.getBlockContent(blockId)
82          prediction = dbms.getBlockPrediction(blockId)
83          try:
84              test_file_content = dbms.project.get_test_content ('block_
    ' + str(blockId) + '_test.dart')
85          except Exception as e:
86              test_file_content = ''
87
88          return jsonify({
89              'content': content,
90              'prediction': prediction,
91              'test_file_content': test_file_content,
92          })
93
94      # dont know why this is needed
95      @app.route('/getBlockDetail', methods=['OPTIONS'])
96      def getBlockDetailOptions():
97          print(request.json)
98          print("wrong method")
99          return jsonify({'message': 'Options request'})
100
101     @app.route('/updateBlockPrediction', methods=['POST'])
102     def updateBlockPrediction():
103         if not request.json or not 'git_url' in request.json or not '
    block_id' in request.json or not 'prediction' in request.json:
104             return jsonify({'message': 'Invalid request'})
105         git_url = request.json['git_url']
106         blockId = request.json['block_id']
107         prediction = request.json['prediction']
108
109         dbms = getDBMS(git_url)
110         dbms.updateBlockPrediction(blockId, prediction)
111
112         return jsonify(
113             {
114                 'message': 'Update success!',
115                 'code': 200,
116                 'success': True,
117             }
118         )
119
120     @app.route('/updateBlockPrediction', methods=['OPTIONS'])
121     def updateBlockPredictionOptions():
122         print(request.json)
123         print("wrong method")
124         return jsonify({'message': 'Options request'})
125
126     @app.route('/generateTest', methods=['POST'])
127     def generateTest():
128         try:
129             if not request.json or not 'git_url' in request.json or
    not 'block_id' in request.json:
130                 return jsonify({'message': 'Invalid request'})
131             git_url = request.json['git_url']
132             blockId = request.json['block_id']
133
```

```
134             dbms = getDBMS(git_url)
135             tg = Test_Generator()
136 
137             testFileContent = tg.generate_test_case(
138                 package_name= dbms.project.getName(),
139                 code_location=dbms.getBlockOriginalFile(blockId),
140                 function_name_and_arguments=dbms.getBlockName(blockId)
    ,
141                 prediction=dbms.getBlockPrediction(blockId),
142             )
143             file_name = 'block_' + str(blockId) + '_test.dart'
144 
145             dbms.project.create_test(
146                 filename=file_name,
147                 content=testFileContent,
148                 isOverWrite=True
149             )
150             # validation process
151             run_result, run_error = dbms.project.run_test(file_name)
152             iteration_limit = 5
153             while run_error != '' and iteration_limit > 0:
154                 new_test_content = tg.fix_generated_code(
155                     error_message=run_error,
156                     code_location=dbms.getBlockOriginalFile(blockId),
157                     function_name_and_arguments=dbms.getBlockName(
    blockId),
158                     prediction=dbms.getBlockPrediction(blockId),
159                 )
160                 dbms.project.create_test(
161                     filename=file_name,
162                     content=new_test_content,
163                     isOverWrite=True
164                 )
165                 run_result, run_error = dbms.project.run_test(
    file_name)
166                 iteration_limit -= 1
167 
168             return jsonify(
169                 {
170                     'message': 'Test generation success!',
171                     'code': 200,
172                     'success': True,
173                     'test_file_content': testFileContent,
174                 }
175             )
176         except Exception as e:
177             return jsonify({'message': str(e)})
178 
179     @app.route('/generateTest', methods=['OPTIONS'])
180     def generateTestOptions():
181         print(request.json)
182         print("wrong method")
183         return jsonify({'message': 'Options request'})
184 
185     if __name__ == '__main__':
186         app.run(debug=True)
```

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

```
1 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

```

```
5       def __init__(self, project) -> None:
6           self.project = project
7
8           # print(self._isDBinit())
9           if not self._isDBinit():
10              self._initDB()
11
12          # print(self._isProjectExistInDB())
13          if not self._isProjectExistInDB():
14              self._insertProject()
15
16          else:
17              # TODO: do something if project already exist
18              pass
19
20      def getJsonDiagram(self) -> dict:
21          """
22          Get the diagram in json format
23          Dict structure:
24          {
25              project: "project_name",
26              blocks: [
27                  {
28                      id: 1,
29                      name: "block_name",
30                      content: "block_content",
31                      prediction: "block_prediction",
32                      type: "block_type"
33                  }
34              ]
35              connections: [
36                  {
37                      head: 1,
38                      tail: 2,
39                      type: "connection_type"
40                  }
41              ]
42          }
43          """
44          # fetch diagram from db
45          blockQuery = Block.getTable().getSelectSQL(fields=['id','name'
    ,'type'],conditions={})
46          blocks = self.execute(blockQuery)
47
48          connectionQuery = Connection.getTable().getSelectSQL(fields
    =[], conditions={})
49          connections = self.execute(connectionQuery)
50
51          # print(blocks)
52          # print(connections)
53
54          res = {
55              'project': self.project.getName(),
56              'blocks': [],
57              'connections': []
58          }
59          for block in blocks:
60              res['blocks'].append({
61                  'id': block[0],
```

```
62                  'name': block[1],
63                  'type': self._getEnumName('BlockType', block[2]),
64              })
65
66          for connection in connections:
67              res['connections'].append({
68                  'head': connection[1],
69                  'tail': connection[2],
70                  'type': self._getEnumName('ConnectionType', connection
    [3])
71              })
72
73          return res
74
75          pass
76
77      def getBlockName(self, blockId: int) -> str:
78          query = Block.getTable().getSelectSQL(fields=['name'],
    conditions={
79              'id': blockId
80          })
81          res = self.execute(query)
82          return res[0][0]
83
84      def getBlockContent(self, blockId: int) -> str:
85          query = Block.getTable().getSelectSQL(fields=['content'],
    conditions={
86              'id': blockId
87          })
88          res = self.execute(query)
89          return res[0][0]
90
91      def getBlockPrediction(self, blockId: int) -> str:
92          query = Block.getTable().getSelectSQL(fields=['prediction'],
    conditions={
93              'id': blockId
94          })
95          res = self.execute(query)
96          return res[0][0]
97
98      def getBlockOriginalFile(self, blockId: int) -> str:
99          # take blockId as tail, query connection table to get head
100         # backtracking until reach FILE block type and return the
    blockname
101         # print('blockId:', blockId)
102         query = Connection.getTable().getSelectSQL(fields=['head'],
    conditions={
103             'tail': blockId
104         })
105         res = self.execute(query)
106         if len(res) > 0:
107             headId = res[0][0]
108             query = Block.getTable().getSelectSQL(fields=['name', '
    type'], conditions={
109                 'id': headId
110             })
111             res = self.execute(query)
112             if len(res) > 0:
113                 blockType = self._getEnumName('BlockType', res[0][1])
```

```
114                 if blockType == 'File':
115                     originalFile = res[0][0]
116                     # exclude lib/
117                     originalFile = originalFile.split('lib/')[1]
118                     return originalFile
119                 else:
120                     return self.getBlockOriginalFile(headId)
121         pass
122 
123     def updateBlockPrediction(self, blockId: int, prediction: str) ->
    None:
124         query = Block.getTable().getUpdateSQL(
125             values={
126                 'prediction': self._handldApostropheString(prediction)
127             },
128             conditions={
129                 'id': blockId
130             }
131         )
132         self.execute(query)
133 
134         pass
135 
136     def _connect(self):
137         self.connection = mysql.connector.connect(
138             host='localhost',
139             user='root',
140             password='1234',
141             database='test_genie'
142         )
143         self.cursor = self.connection.cursor(buffered=True)
144 
145     def _close(self):
146         self.cursor.close()
147         self.connection.close()
148 
149     def _isDBinit(self):
150         query = 'SHOW TABLES'
151         res = self.execute(query)
152         return len(res) >= self._numberOfTables
153 
154     def execute(self, query) -> list:
155         self._connect()
156 
157         if type(query) == str:
158             self.cursor.execute(query)
159         else:
160             for q in query:
161                 self.cursor.execute(q)
162         self.connection.commit()
163 
164         self._close()
165         return self.cursor.fetchall() # type: ignore
166 
167     def _initDB(self):
168         projectQuery = self.project.getTable().getCreateSQL()
169         self.execute(projectQuery)
170 
171         blockCreateQuery = Block.getTable().getCreateSQL()
```

```
172         self.execute(blockCreateQuery)
173 
174         connectionQuery = Connection.getTable().getCreateSQL()
175         self.execute(connectionQuery)
176         self._insertEnumDB()
177 
178     def _insertEnumDB(self):
179 
180         blockTypeCreateQuery = BlockType.getTable().getCreateSQL()
181         # print(blockTypeQuery)
182         self.execute(blockTypeCreateQuery)
183 
184         blockTypeInsertQuery = BlockType.getInsertQuery()
185         # print(blockTypeInsertQuery)
186         self.execute(blockTypeInsertQuery)
187 
188         connectionTypeCreateQuery = ConnectionType.getTable().
    getCreateSQL()
189         self.execute(connectionTypeCreateQuery)
190 
191         connectionTypeInsertQuery = ConnectionType.getInsertQuery()
192         # print(connectionTypeInsertQuery)
193         self.execute(connectionTypeInsertQuery)
194 
195     def _isProjectExistInDB(self):
196         query = self.project.getTable().getSelectSQL(fields=['name'],
    conditions={
197             'name': self.project.getName()
198         })
199         # print(query)
200         res = self.execute(query)
201         return len(res) > 0
202 
203     def _insertProject(self):
204         # print('Inserting project')
205         # project table insert
206         query = self.project.getTable().getInsertSQL({
207             'name': self.project.getName(),
208             'directory': self.project.getPath()
209         })
210         # print(query)
211         self.execute(query)
212         # diagram insert
213         diagram = DependencyDiagram(self.project)
214         # not sure if this is needed
215         self.diagram = diagram
216 
217         blocks = diagram.blocks
218         connections = diagram.connections
219         self._mapBlocksIntoDB(blocks)
220         self._mapConnectionsIntoDB(connections)
221 
222     def _mapBlocksIntoDB(self, blocks: list):
223         for block in blocks:
224             # TODO: handle apostrophe in content
225             # map into block table
226             query = Block.getTable().getInsertSQL({
227                 'name': self._handldApostropheString(block.name),
228                 'content': self._handldApostropheString(block.content)
```

```
    ,
229                 'prediction': self._handldApostropheString(block.
    prediction),
230                 'type': self._getEnumId('BlockType', block.type)
231             })
232             self.execute(query)
233         pass
234     def _mapConnectionsIntoDB(self, connections: list):
235         for connection in connections:
236             # map into connection table
237             query = Connection.getTable().getInsertSQL({
238                 'head': self._getBlockId(connection.head),
239                 'tail': self._getBlockId(connection.tail),
240                 'type': self._getEnumId('ConnectionType', connection.
    type)
241             })
242             self.execute(query)
243         pass
244     def _getBlockId(self, block) -> int:
245         table = Block.getTable()
246         query = table.getSelectSQL(fields=['id'], conditions={
247             'name': self._handldApostropheString(block.name),
248             'content': self._handldApostropheString(block.content),
249             'prediction': self._handldApostropheString(block.
    prediction),
250             'type': self._getEnumId('BlockType', block.type)
251         })
252         res = self.execute(query)
253         return res[0][0]
254     def _getEnumId(self, enum, enumName: str) -> int:
255         # base on enumname to get blocktype or connectiontype id
256         if enum in globals():
257             enumClass = globals()[enum]
258             table = enumClass.getTable()
259             query = table.getSelectSQL(fields=['id'], conditions={
260                 'name': enumName
261             })
262             res = self.execute(query)
263             return res[0][0]
264         return 0
265     def _getEnumName(self, enum, enumId: int) -> str:
266         if enum in globals():
267             enumClass = globals()[enum]
268             table = enumClass.getTable()
269             query = table.getSelectSQL(fields=['name'], conditions={
270                 'id': enumId
271             })
272             res = self.execute(query)
273             return res[0][0]
274         return ''
275     def _handldApostropheString(self, string: str) -> str:
276         return string.replace("'", "''")
277         pass
```

Listing A.21: DBMS class.