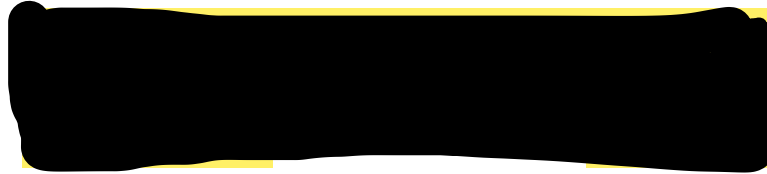


# RefactoAI

Revamping Code Excellence through AI-Driven Code Refactoring

Project Team



Session 2021-2025



Department of Computer Science

National University of Computer and Emerging Sciences  
Islamabad, Pakistan

June, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement . . . . .	1
1.2	Scope . . . . .	2
1.3	Modules . . . . .	2
1.3.1	<b>Module 1: Data Collection and Analysis</b> . . . . .	2
1.3.2	<b>Module 2: Requirement Engineering</b> . . . . .	3
1.3.3	<b>Module 3: Code Refactoring and Regeneration Utilizing DeepSeek-R1-Distill-Qwen-32B</b> . . . . .	3
1.3.4	<b>Module 4: Development of Front-End, Testing, and Performance Optimization</b> . . . . .	3
1.4	User Classes and Characteristics . . . . .	4
<b>2</b>	<b>Project Requirements</b>	<b>5</b>
2.1	Use-case/Event Response Table/Storyboarding . . . . .	5
2.2	Functional Requirements . . . . .	7
2.2.1	Module 1: Code Upload and Validation . . . . .	7
2.2.2	Module 2: User Story Integration . . . . .	8
2.2.3	Module 3: Refactoring and Regeneration . . . . .	8
2.2.4	Module 4: Testing and Reporting . . . . .	8
2.3	Non-Functional Requirements . . . . .	8
2.3.1	Reliability . . . . .	8
2.3.2	Usability . . . . .	9
2.3.3	Performance . . . . .	9
2.3.4	Security . . . . .	9
2.4	Domain Model . . . . .	11
<b>3</b>	<b>System Overview</b>	<b>13</b>
3.1	Architectural Design . . . . .	13
3.2	Design Models . . . . .	15
3.3	Data Design . . . . .	19
3.3.1	Major Data Entities and Structures . . . . .	19
3.3.2	Data Flow and Processing Stages . . . . .	20
3.3.3	Data Storage and Organization . . . . .	21

3.3.4	Conclusion . . . . .	21
<b>4</b>	<b>Implementation and Testing</b>	<b>23</b>
4.1	General Description . . . . .	23
4.2	Algorithm Design . . . . .	24
4.2.1	Module 1: Analysis . . . . .	24
4.2.1.1	Pseudocode . . . . .	24
4.2.2	Module 2: Requirement Engineering . . . . .	25
4.2.2.1	Pseudocode . . . . .	25
4.2.3	Module 3: Code Refactoring Utilizing DeepSeek-R1-Distill-Qwen-32B . . . . .	25
4.2.3.1	Pseudocode . . . . .	26
4.3	External APIs/SDKs . . . . .	26
4.4	Testing Details . . . . .	28
4.4.1	Integration Testing . . . . .	28
4.4.1.1	Example of Integration Testing . . . . .	28
4.4.2	System Testing . . . . .	28
4.4.2.1	Example of System Testing . . . . .	29
4.4.3	React Application Testing . . . . .	29
4.4.3.1	Test Input . . . . .	29
4.4.3.2	Code Quality Analysis . . . . .	29
4.4.3.3	Generated User Stories . . . . .	30
4.4.3.4	Modified Code . . . . .	30
4.4.4	Requirement Report Generation . . . . .	33
4.4.4.1	Test Results . . . . .	35
4.4.4.2	User Experience . . . . .	35
4.4.4.3	Some More Testing Images . . . . .	35
4.5	Current Iteration Summary . . . . .	41
4.6	Conclusion . . . . .	41
	<b>References</b>	<b>43</b>

# List of Figures

2.1	Use Case Diagram . . . . .	6
2.2	Domain Model Diagram . . . . .	11
3.1	Architectural Diagram . . . . .	14
3.2	Activity Diagram . . . . .	15
3.3	State Sequence Diagram (SSD) . . . . .	16
3.4	State Machine Diagram . . . . .	17
3.5	Data Flow Diagram . . . . .	18
4.1	Frontend Interface: Python File Upload and Analysis . . . . .	35
4.2	Frontend Interface: User Stories Generation . . . . .	36
4.3	Frontend Interface: Displaying Results and Modified Code . . . . .	36
4.4	Requirement Report . . . . .	37
4.5	Libraries-Only File . . . . .	38
4.6	Empty File Uploaded . . . . .	38
4.7	Syntax Error Detected . . . . .	39
4.8	Missing Requirements Prompt . . . . .	39
4.9	Modified Code Output . . . . .	40

# List of Tables

1.1	User Classes and Characteristics for Fully Automated RefactoAI . . . . .	4
2.1	Use-Case Table for Fully Automated RefactoAI . . . . .	7
2.2	Product, Organizational, and External Requirements for Fully Automated RefactoAI . . . . .	10
4.1	External APIs and SDKs Used in RefactoAI . . . . .	27
4.2	Requirement Report . . . . .	34

# Chapter 1

## Introduction

This document of project proposal is aimed at outlining the objectives, scope, and implementation plan for the project "RefactoAI– Revamping Code Excellence through AI Driven Refactoring" This document gives in detail general view, background of the system, the problems that it intends to handle, and the solution that it sets forward. This project aims to come up with a web-based application that will facilitate the automation of refactoring Python code and regenerating the code using advanced large language models based on the requirements of the user.

### 1.1 Problem Statement

Under the condition of continuous change in modern software development, quality code is the essential underpinning to sustain applications over the long term [3]. The paper further considers that traditional code refactoring processes are of much manual effort, time-consuming, and thus are very sensitive to human error [5]. Evidently, from previous studies, it was shown that such manual efforts result in a lot of inefficiencies, increasing the rates of error and lowering both code quality and developer productivity levels [2]. This is due to the fact that a lot of time has to be spent on addressing quality, alignment, performance, and other issues in the code [1]. Research also suggests that bugs or other problems related to refactoring can increase the complexity of the development cycle [4]. Therefore, it is mandatory to come up with advanced automated solutions based on newly proposed techniques [6].

This project builds an automated solution that leverages advanced LLMs for refactoring and regenerating Python code from requirements. The purpose of the system is to facilitate the refactoring process itself, reducing the manual effort and enhancing the quality of the code. Receiving the requirements at the input of the refactoring process means that the system ensures results will be user-satisfying and project-expectation matching. It is

expected that this would save time, prevent errors, and improve the maintainability of the code in a whole, both in meeting practical developer needs and the theoretical challenges stated in existing research reports [4] [5] .

## 1.2 Scope

The project "RefactoAI—Revamping Code Excellence through AI-Driven Code Refactoring" is designed to implement a web-based application using advanced Large Language Models (LLMs) such as DeepSeek RI Distill Quen 2.5 and Microsoft Phi-4 Mini for automating Python code refactoring. The core functionalities include analyzing existing Python scripts, collecting user-defined requirements, converting these into structured user stories using an LLM, and refactoring the code to align with the defined goals.

To evaluate requirement fulfillment, the system uses a custom transformer model that parses the regenerated code line by line. It calculates semantic similarity between requirement keywords and code segments using a dynamic threshold. The final report clearly indicates whether each requirement has been met, and if so, identifies the specific lines of code that correspond to it.

This end-to-end automation enhances code quality, readability, and maintainability, providing an efficient and intelligent solution exclusively tailored for Python development.

## 1.3 Modules

### 1.3.1 Module 1: Data Collection and Analysis

This module involves collecting and analyzing Python code that requires enhancement.

1. **Datasets Required:** Collect data from the following sources:
  - GeeksforGeeks Python projects
  - CopyAssignment Python programming projects
  - Project Euler solutions from GitHub repository
2. **Executing Tests Over the Script:** Execute test cases on the provided Python script by the user to identify defects, including issues related to alignment, indentation, or coding standards.
3. **Refactor the Code:** Clean up the code by improving alignment, removing extra indentations, and enhancing quality and best practices.

4. **Feedback & Suggestions:** Provide feedback on found errors in the code. Include detailed errors and suggestions for user code improvement.
5. **Design-Figma:** Develop Figma designs for the web application during the design phase.

### 1.3.2 Module 2: Requirement Engineering

This module involves decomposing user requirements into technical user stories using an open-source LLM.

1. **Gather Requirements:** Collect requirements from clients/users.
2. **Analyze and Synthesize:** Use an open-source LLM to analyze and synthesize the collected requirements into user stories in natural language.

### 1.3.3 Module 3: Code Refactoring and Regeneration Utilizing DeepSeek-R1-Distill-Qwen-32B

This module involves refactoring and regenerating Python code based on recognized user stories.

1. **Convert User Stories:** Convert user stories into code requirements.
2. **Embed Use Cases:** Integrate the use cases into the original source code using DeepSeek-R1-Distill-Qwen-32B.
3. **Verify Code:** Ensure that the refactored code meets all the requirements outlined in the user stories.

### 1.3.4 Module 4: Development of Front-End, Testing, and Performance Optimization

This module involves developing the user interface, optimizing performance, and performing final validation of the refactored code.

1. **Develop Web-Based Application Interface:** Create the user interface for the web application.
2. **Add Testing Tools:** Implement tools for automatic error detection and resolution.



3. **Optimize Refactored Code:** Maximize the performance of the refactored code.

4. **Final Testing and Requirement Validation:**

- A custom transformer-based model is used to semantically match user stories against the final regenerated code.
- For each user story, the system parses the code line by line, calculates similarity scores, and compares them to a dynamic threshold.
- A report is then generated indicating whether each requirement was met. If matched, the corresponding line(s) of code are shown alongside the requirement.
- **Reference Output:** As shown in the sample below, the report maps user stories to functions or lines of code such as:
  - REQ1 → Function: `calculate_stats` (Line 71)
  - REQ2 → Line 106: `pitch = calculate_pitch(denoised_audio, sample_rate)`
  - REQ3 → Function: `gender_classification` (Line 94)

## 1.4 User Classes and Characteristics

User Classes and Characteristics of RefactoAI

User Class	Description
User	A User is any individual who uses the system to upload Python code for refactoring and analysis. This could be a software engineer, developer, or a person without deep technical expertise. Users are expected to provide code and refactoring requirements. The system will guide them through the process of specifying their needs for code optimization and testing.
System (Automated)	The system performs all backend processes autonomously. It uses large language models (LLMs) to refactor and regenerate Python code based on user inputs. The system handles code validation, error detection, automated testing, and performance optimization without any human intervention. It generates reports mapping user requirements to the refactored code.

Table 1.1: User Classes and Characteristics for Fully Automated RefactoAI

# Chapter 2

## Project Requirements

This chapter describes the functional and non-functional requirements of the project.

### 2.1 Use-case/Event Response Table/Storyboarding

- The **Use Case Diagram** for the RefactoAI system illustrates the complete workflow of how the user interacts with the system to refactor Python code using Large Language Models (LLMs). The process starts with the User uploading a Python file, which triggers the system to check the Python file for syntax errors. If the file contains errors (shown as a failure path), the system stops and requires the user to correct and re-upload the file. If the file passes validation, the system gathers user requirements and proceeds to generate user stories using an LLM. Based on the generated user stories, the system continues with code generation or refactoring through another LLM, and subsequently, the refactored code undergoes testing. If the generated code passes testing, a new Python file is created and returned to the user. If the code fails testing, it returns to the refactoring phase for further adjustments. In this diagram, the «include» relationship shows mandatory steps in the process, such as checking the file after it is uploaded. The «extend» relationship represents optional or conditional steps, like gathering user requirements extending the generation of user stories.

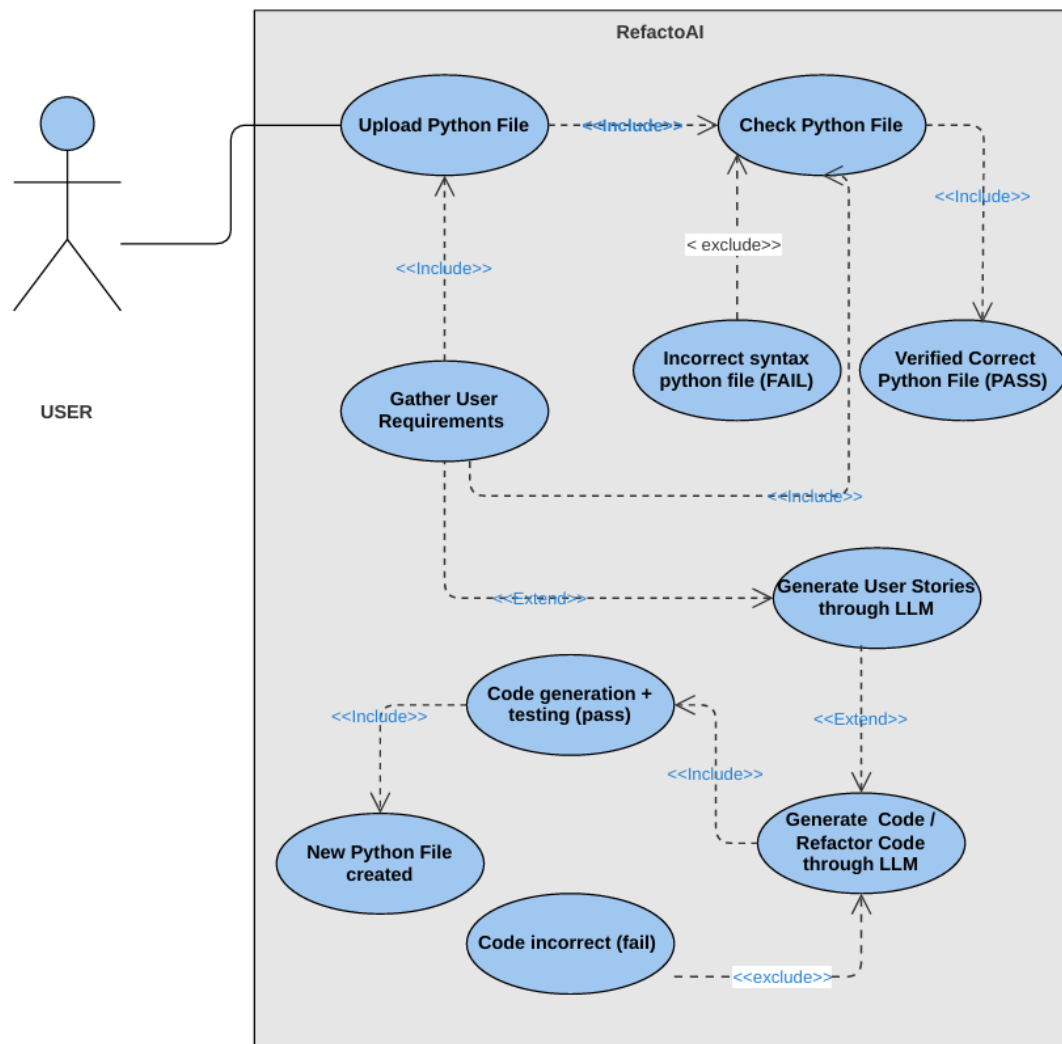


Figure 2.1: Use Case Diagram

Use Case	Actor	Description	System Response
Upload Code	User	The user uploads a Python script for refactoring.	The system automatically validates the code for errors (e.g., syntax, alignment) and prepares it for refactoring.
Input Requirements	User	The user specifies new requirements or user stories for the refactoring process.	The system processes these requirements using LLMs, refactors the code, and incorporates user-defined changes.
View Refactored Code	User	The user views the refactored code and its improvements.	The system displays the refactored code, including detailed changes and optimization suggestions.
Automated Testing	System	The system runs automated tests on the refactored code.	The system validates the code, ensuring it meets user stories and project requirements, and generates a report automatically.
Generate Report	System	The system generates a report on the refactoring process and the code's alignment with user requirements.	The system displays the report for the user, detailing the refactoring steps, test results, and final code status.

Table 2.1: Use-Case Table for Fully Automated RefactoAI

## 2.2 Functional Requirements

Below are the functional requirements of RefactoAI which will tell what the system is suppose to achieve.

### 2.2.1 Module 1: Code Upload and Validation

Following are the requirements for Module 1:

1. FR1: The system shall allow users to upload Python scripts for analysis.
2. FR2: The system shall validate the uploaded code for errors (syntax, indentation, etc.) before refactoring.
  - FR2.1: Upon error, the system shall ask the user to upload a new file.
  - FR2.2: If no errors are found, the system shall continue and move to the next step.

### **2.2.2 Module 2: User Story Integration**

Following are the requirements for Module 2:

1. FR1: The system shall allow users to input requirements for the code.
2. FR2: The system shall analyze these requirements and translate them into user stories using open-source LLMs.

### **2.2.3 Module 3: Refactoring and Regeneration**

Following are the requirements for Module 3:

1. FR1: The system shall refactor the uploaded code to improve its quality, performance, and maintainability.
2. FR2: The system shall regenerate the code by embedding the new user requirements into the existing code base.

### **2.2.4 Module 4: Testing and Reporting**

Following are the requirements for Module 4:

1. FR1: The system shall validate the refactored code using pyRat or other testing tools.
2. FR2: The system shall generate a detailed report mapping user stories to the refactored code.

## **2.3 Non-Functional Requirements**

This section specifies non-functional requirements.

### **2.3.1 Reliability**

- Requirement: The system must operate without failure 90.5% of the time.
- Measurement: Mean time between failures (MTBF) should be at least 100 hours. The system must log any failures and automatically restart the refactoring process in case of an interruption.

### 2.3.2 Usability

- **Requirement:** The system should allow users to upload and refactor code with no more than 3 steps.
- **Example:** The interface shall be intuitive, allowing users to complete the refactoring process within 5 minutes of interaction.

### 2.3.3 Performance

- **Requirement:** The system must refactor the uploaded code within 5 seconds for small scripts (< 200 lines) and under 15 seconds for large scripts (up to 1000 lines).
- **Measurement:** Performance must be measured during peak hours when multiple users are refactoring code simultaneously.

### 2.3.4 Security

- **Requirement:** All user-uploaded code must be encrypted while stored in the system to protect intellectual property.
- **Measurement:** The system should resist unauthorized access with an intrusion detection mechanism that alerts developers of any suspicious activities.

<b>Requirement Category</b>	<b>Subheading</b>	<b>Description</b>
<b>Product Requirements</b>	Performance	The system must refactor code within 10 seconds for small scripts (<500 lines) and under 30 seconds for larger scripts (up to 2000 lines). Performance should be optimal under peak loads.
	Usability	The interface should be user-friendly, allowing users to upload code and view results with minimal steps (within 2 interactions).
	Reliability	The system should have an MTBF (Mean Time Between Failures) of over 300 hours and log any failures automatically. It must operate without failure 90.9% of the time.
	Scalability	The system should scale to support up to 100 simultaneous users and 500 code uploads per minute without performance degradation.
<b>External Requirements</b>	Compliance	The system must comply with data protection laws (e.g., GDPR) and ensure the security and confidentiality of user-uploaded code and data.
	Security	All uploaded code and data must be encrypted and stored securely. Unauthorized access attempts must be blocked, and incidents should be logged.
	External System Integration	The system must support integration with third-party tools (like LLM APIs) for code refactoring and testing functionalities (in the future).

Table 2.2: Product, Organizational, and External Requirements for Fully Automated RefactoAI

## 2.4 Domain Model

A **domain model** represents the key entities, their attributes, and relationships in a system, describing how they interact to fulfill the system's purpose. It provides a conceptual view of the real-world objects that exist in the system and the relationships between them. In the case of RefactoAI, the domain model would include entities such as the User, Python Script, LLM Models (for user story generation and code refactoring), requirements, and the testing tools. These entities work together to automate the code refactoring process from input (the Python script) to output (the refactored and tested code), based on the user's requirements and LLM driven processing.

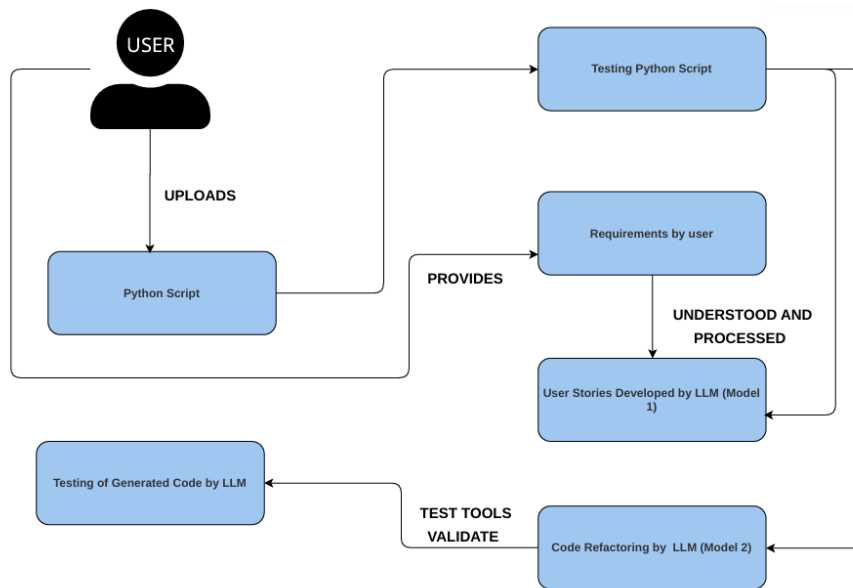


Figure 2.2: Domain Model Diagram



# Chapter 3

## System Overview

This project – RefactoAI – Revamping Code Excellence through AI-Driven Refactoring would be such a system that regenerates and improves the Python code using large language models. In this system, users can upload Python scripts, specify the requirements needed to be embedded and obtain refactored code with no human input. The focus is mainly on making the tedious work of code refactoring which is however manual and often full of errors, accurate and entirely automated in order to improve the quality, maintainability, and performance of the code with less interaction from the developer.

### 3.1 Architectural Design

- **User Interface (UI) Layer:**
  - The user uploads a Python file (.py) through the interface, initiating the refactoring process.
- **Application Layer (Logic):**
  - **Script Validation:** Ensures the uploaded script adheres to syntax rules.
  - **Generate User Stories:** User requirements are processed with an LLM to generate user stories stored in memory.
  - **Refactor Code:** Another LLM refactors the code based on a pre-trained dataset.
  - **Testing:** The refactored code is tested for performance and correctness.
- **File System Layer:**
  - Files (temporarily stored in a folder) hold the refactored code during processing.

- **Trained Data Layer:**

- The pre-trained dataset of 2 million Python scripts informs the LLM during code refactoring.

- **Conclusion:**

- The system uses file-based temporary storage to minimize computational overhead and improve efficiency during processing.

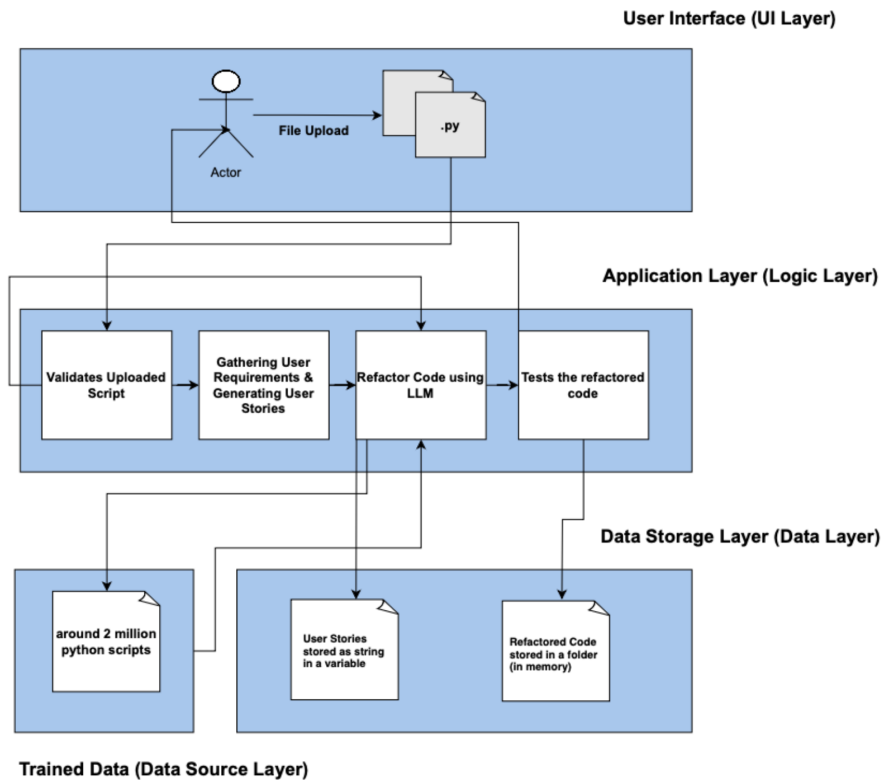


Figure 3.1: Architectural Diagram

## 3.2 Design Models

### Design Models for Procedural Approach

- This **activity diagram** represents the workflow for the RefactoAI system, starting with the User uploading a Python file for refactoring. After the upload, the file undergoes validation to check for errors. If errors are found, the process moves to Requirement Specification, where the user can correct the requirements. Once the file passes validation, the system moves to User Stories generation using an LLM. The generated use cases or stories are then passed to the Code Generation/Refactoring phase, where another LLM refactors the Python code. After refactoring, the code is subjected to testing using different tools. If the code passes the testing phase, it is marked as verified code. If it fails the test, the process loops back to the Code Generation/Refactoring step, allowing for iterative improvements based on the test results.

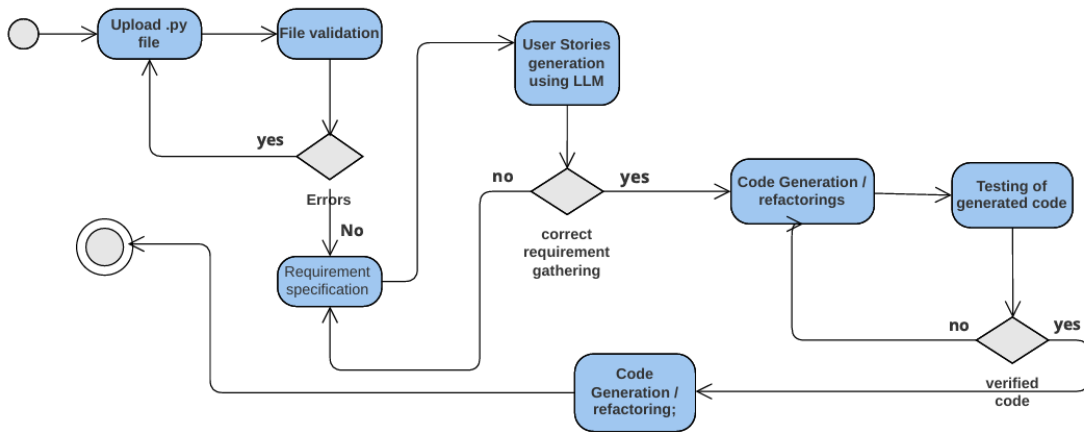


Figure 3.2: Activity Diagram

- The **System Sequence Diagram (SSD)** for RefactoAI outlines the interaction flow between the User and the System during the code refactoring process. The process starts with the User uploading a Python file, which is then validated by the system. If the validation returns errors, the system requests the user to correct the file and re-upload it. Once the file is valid, the user provides specific requirements, and the system uses LLMs to generate user stories. The generated user stories are then returned to the user. Following this, the system proceeds to the code refactoring phase using LLM, where it refactors or generates new code. The refactored code is tested, and if the tests pass, the system sends the verified code back to the user as a newly refactored Python file. The diagram illustrates this entire interaction and the flow of tasks from upload to the final delivery of the refactored Python file.

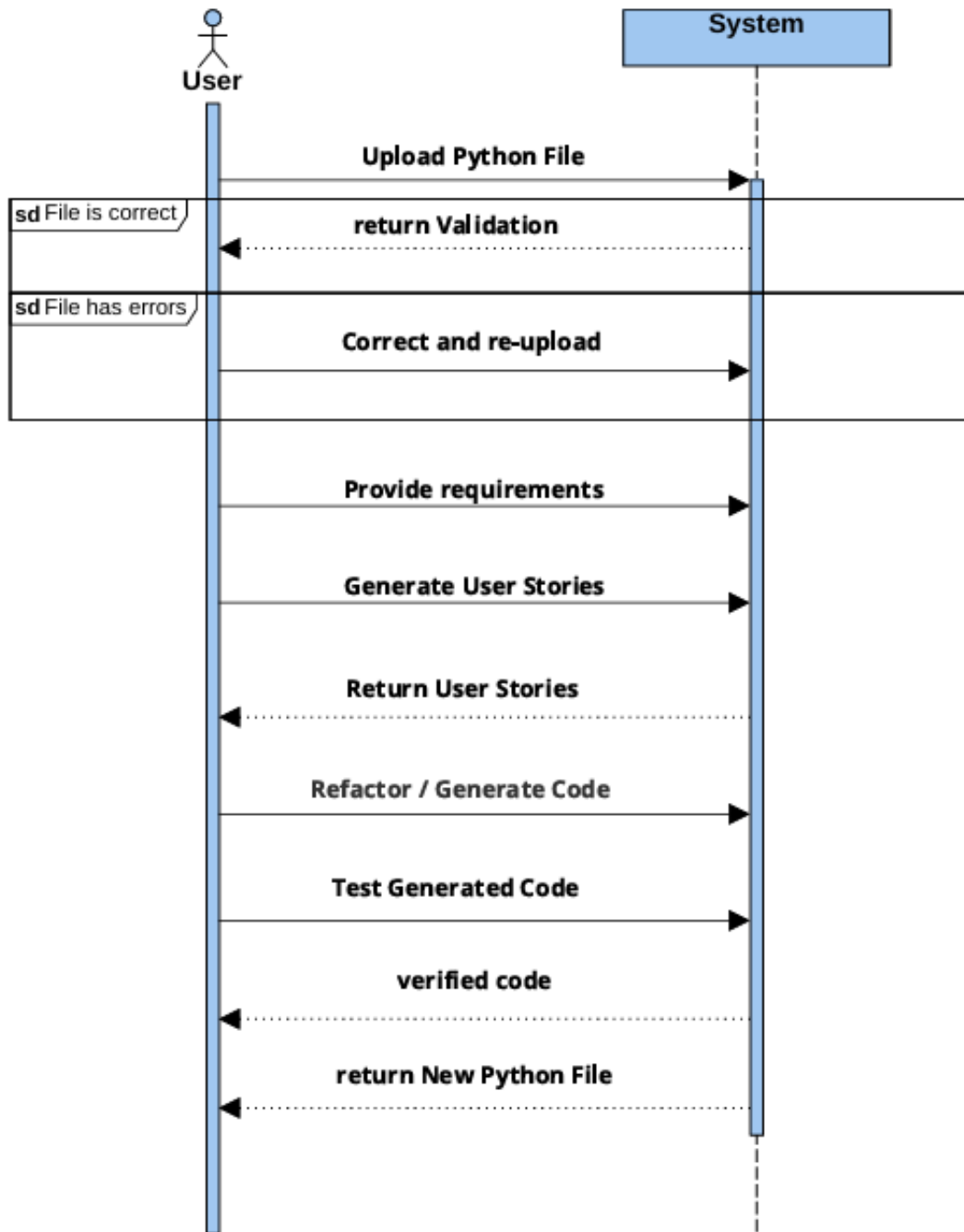


Figure 3.3: State Sequence Diagram (SSD)

- The **State Machine Diagram** for RefactoAI illustrates the progression of a Python script through various states, from upload to final delivery, with temporary file storage utilized at multiple stages to optimize the process. The system begins when the User Uploads the Python Script (Uploaded state). The script is then Validated for errors; if it fails, the user must upload a corrected version, repeating the cycle. If the script passes validation, the system Gathers Requirements, which are stored temporarily in the User Stories Storage (Files). After that, User The LLM generates stories, and the process moves to Code Refactoring using a pre trained dataset of Python scripts, temporarily storing the refactored code in Refactor Code Storage (Files). The refactored code is then Tested, and if it passes, the system creates a new Python file in the Finalized state, delivering it back to the user. If the test fails, the system loops back to refactor the code again. By using files temporarily, the system minimizes computational costs that would otherwise arise from a database. Temporary file storage allows for faster access and processing without the overhead of long-term data management, making the system efficient and scalable. Each state transition is triggered by successful or failed validations, user input, and LLM processing events.

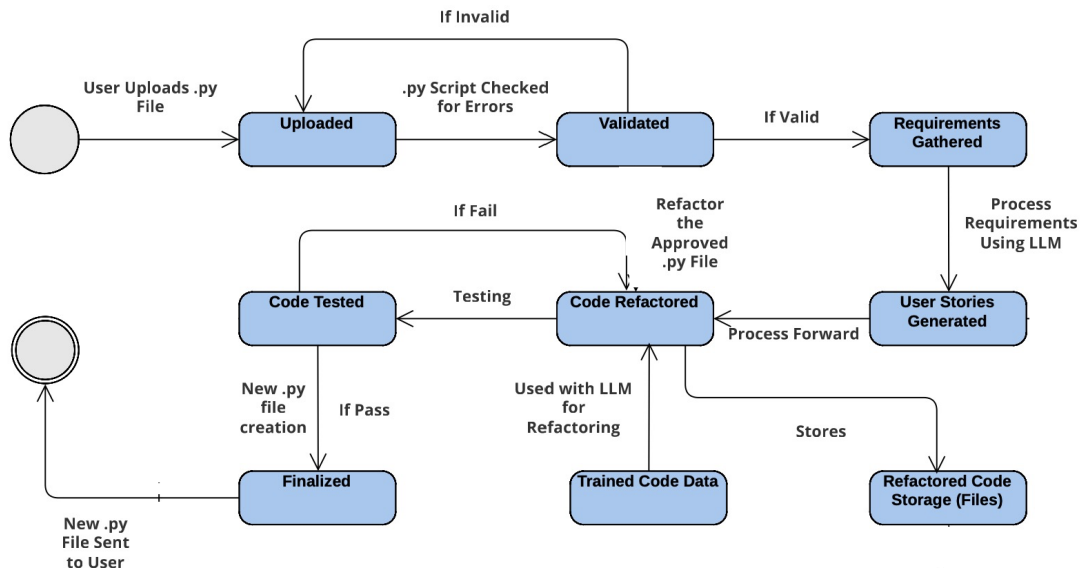


Figure 3.4: State Machine Diagram

- The **Data Flow Diagram (DFD)** for RefactoAI illustrates the systematic flow of data through various components of the system, highlighting how user inputs are transformed into outputs. The process begins with the user uploading a Python script, which triggers subsequent actions within the system. Initially, the uploaded script undergoes validation in the "Validate Python Script" phase, where it is checked

for correctness, and the user may be prompted for additional requirements. If the script is validated successfully, the system analyzes the requirements and converts them into user stories using a Language Model (LLM1). These user stories, along with the original script, are then processed in the "Refactor Code" phase, where a second Language Model (LLM2) is employed for refactoring and regeneration, enhancing the code's structure and efficiency. The refactored code is temporarily stored in the "Refactor Code Storage (Folder)" for further testing. In the "Test Refactored Code" phase, the system verifies the functionality of the refactored code. After successful testing, the final version of the code is sent back to the user, completing the data flow. This DFD emphasizes a file-based system for data storage, chosen for its efficiency and simplicity, as it minimizes overhead compared to traditional databases, ensuring smooth transitions of necessary data between processes.

Overall, it provides a clear representation of how data flows from the user through various stages of processing, storage, and output, optimizing the refactoring process within the RefactoAI system.

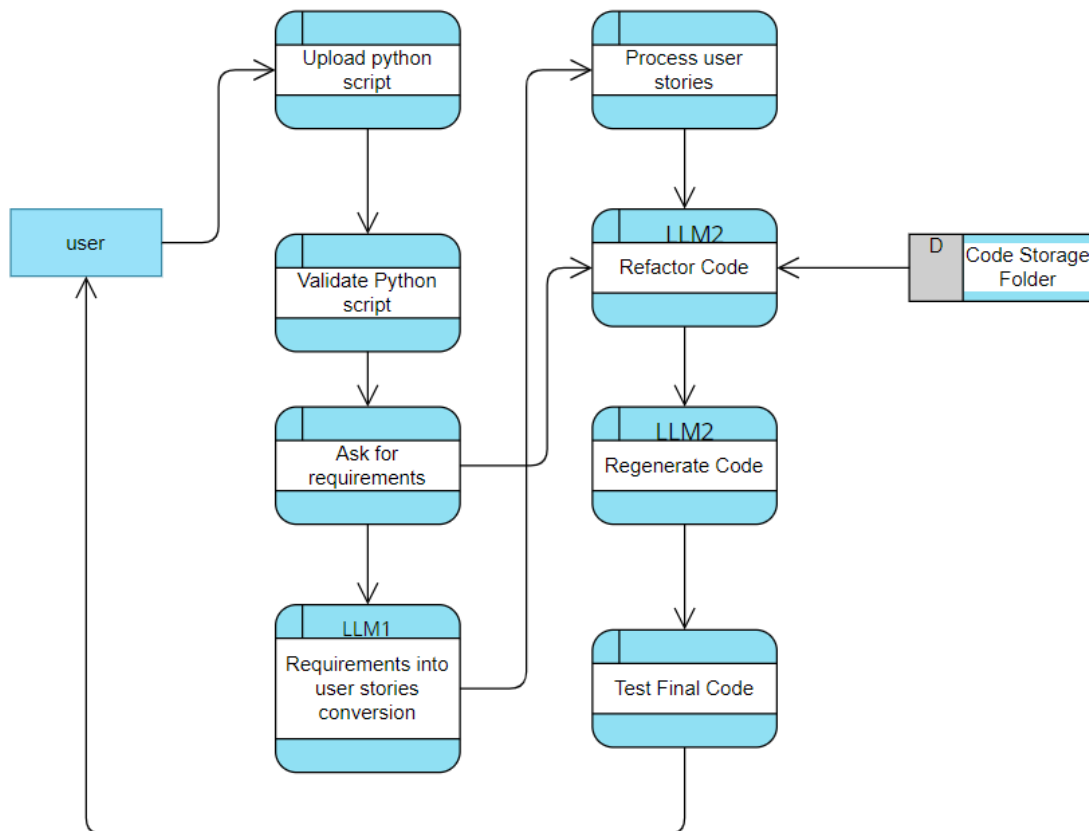


Figure 3.5: Data Flow Diagram

## 3.3 Data Design

The data design of RefactoAI outlines how data flows through the system, how it is processed, stored, and organized. The core entities involve the user-uploaded Python file, LLM-generated user stories, refactored and regenerating code, and the final report that links user requirements to specific code changes.

### 3.3.1 Major Data Entities and Structures

- **User-Uploaded Python File:**
  - **Data Structure:** The Python file is treated as a *file object*. The whole file is read into the system for analysis and refactoring.
  - **Storage:** The file is temporarily stored in a directory `/user_uploads/` until analysis and refactoring are complete.
  - **Processing:** The system runs tests on the file, including syntax checks, PEP8 compliance, and code quality analysis. If errors are found, the user is asked to upload a new file. If no errors are found, the system provides suggestions and a score for the code.
- **Test Results and Suggestions:**
  - **Data Structure:** Test results from syntax checks, PEP8 compliance, and complexity analysis are stored as *text output*.
  - **Storage:** These results are temporarily stored in memory and displayed to the user.
  - **Processing:** If no errors are found, suggestions and a score are generated. If errors are detected, the system prompts the user to upload a new file.
- **User Requirements:**
  - **Data Structure:** User requirements are entered as *text* and stored as a *list of strings*.
  - **Storage:** These requirements are stored temporarily in memory and used as input for LLM1.
  - **Processing:** The requirements are sent to **LLM1**, which converts them into user stories.
- **User Stories (LLM1 Output):**
  - **Data Structure:** The output from **LLM1** is a set of *text-based user stories*.

- **Storage:** These user stories are stored temporarily in memory and passed to the next stage for refactoring.
- **Processing:** The system passes the user stories to **LLM2**, which regenerates the code based on the user stories.
- **Refactored and Regenerated Code (LLM2 Output):**
  - **Data Structure:** The refactored and regenerated code is saved as a *file object*.
  - **Storage:** The new Python file is saved in the `/user_uploads/` directory.
  - **Processing:** **LLM2** refactors and regenerates the code based on the user stories. The final code is then tested to ensure no errors are introduced.
- **Final Report (Requirement-to-Code Mapping):**
  - **Data Structure:** The final report is structured as a *text file* or *JSON object* and links each part of the new code to corresponding user requirements.
  - **Storage:** The report is stored and made available for the user to view or download.
  - **Processing:** The report shows how the user requirements map to the specific parts of the refactored code.

### 3.3.2 Data Flow and Processing Stages

1. **User Uploads Python Code:** The user uploads the Python file, which is stored in a directory on the server. The system runs tests (syntax, PEP8, quality) on the file. If errors are detected, the user is prompted to upload a new file. If no errors are found, suggestions and a score are shown.
2. **User Inputs New Requirements:** Once the uploaded code passes the tests, the user inputs new requirements. These requirements are stored as a list of strings.
3. **LLM1: Convert Requirements to User Stories:** The user's requirements are sent to **LLM1**, which converts them into user stories. The stories are stored temporarily and passed to the next stage for refactoring.
4. **LLM2: Refactor Code Based on User Stories:** Both the original Python file and the user stories are passed to **LLM2**, which refactors and regenerates the code. The refactored code is saved in the `/user_uploads/` directory.
5. **Generate Requirement-to-Code Mapping Report:** The system generates a report that links user requirements to specific parts of the refactored code.



6. **Display Refactored Code and Report:** The refactored Python code and the final report are shown to the user, allowing them to review the changes and understand how their requirements were implemented.

### 3.3.3 Data Storage and Organization

- **File System:** Uploaded Python files and refactored versions are stored in a folder `/user_uploads/`.
- **In-Memory Data:** Temporary data, such as user requirements, LLM outputs (user stories and refactored code), and test results, are stored in memory during processing. The data is discarded after the final output is generated.
- **Report Generation:** The final report shows how user requirements are linked to specific parts of the refactored code. This report is saved in a structured format (text or JSON) and made available for download.

### 3.3.4 Conclusion

In RefactoAI, the data design transforms user inputs (Python files and requirements) into outputs (refactored code and reports) through various stages. Data is primarily stored in the file system, with intermediate steps processed in memory for efficiency. The final report provides a clear link between the user's requirements and the changes made in the code, allowing for transparency and understanding of the refactoring process.

# Chapter 4

## Implementation and Testing

This chapter provides a comprehensive overview of the implementation and testing phases of the RefactoAI project up to the current iteration. It outlines the functionality, context, and design of the system, detailing the algorithms employed, the external APIs and SDKs integrated, and the testing methodologies adopted to ensure the system operates as intended.

### 4.1 General Description

RefactoAI is designed as a web-based application that automates the refactoring and regeneration of Python code using advanced Large Language Models (LLMs) such as Phi-4-Mini-Instruct and DeepSeek-R1-Distill-Qwen-32B. The system consists of a backend developed with FastAPI and a frontend built using React. The primary functionalities include:

- **Code Upload and Validation:** Users can upload Python scripts, which are then validated for syntax errors and adherence to PEP8 standards.
- **Requirement Specification:** Users can input requirements that are converted into detailed user stories using Phi-4-Mini-Instruct.
- **Code Refactoring and Regeneration:** Based on the user stories, DeepSeek-R1-Distill-Qwen-32B refactors and regenerates the Python code.
- **Report Generation:** A semantic matching module generates a report that checks if each requirement is implemented in the code. It lists whether the requirement is met and highlights the specific lines or functions where the match occurs.
- **User Interface:** The frontend provides an intuitive interface for users to interact with the system, upload files, specify requirements, and view results.

The implementation leverages several third-party libraries and frameworks to enhance functionality and streamline development. The following sections delve into the algorithm design, external APIs/SDKs used, and the testing strategies employed.

## 4.2 Algorithm Design

The RefactoAI system leverages several algorithms to automate the process of code refactoring and regeneration using Large Language Models (LLMs). The core functionalities are distributed across major modules, each utilizing specific algorithms to achieve desired outcomes. Below is a detailed explanation of the algorithms employed in the primary modules of the system.

### 4.2.1 Module 1: Analysis

#### Algorithm: Code Validation and Quality Analysis

**Description:** This algorithm validates the uploaded Python code for syntax errors, adherence to coding standards (PEP8), and overall code quality. It ensures that only error-free code is processed for refactoring.

#### 4.2.1.1 Pseudocode

```
function validate_code(file_path):
    code = read_file(file_path)
    errors = syntax_check(code)
    if errors:
        return {"status": "error", "details": errors}
    pep8_violations = check_pep8(code)
    if pep8_violations:
        return {"status": "warning", "details": pep8_violations}
    quality_score = analyze_code_quality(code)
    return {"status": "success", "score": quality_score}

function syntax_check(code):
    try:
        compile(code, '<string>', 'exec')
        return None
    except SyntaxError as e:
        return str(e)
```

```

function check_pep8(code):
    violations = run_pep8_linter(code)
    return violations

function analyze_code_quality(code):
    metrics = compute_code_metrics(code)
    score = calculate_quality_score(metrics)
    return score

```

## 4.2.2 Module 2: Requirement Engineering

### Algorithm: Requirement to User Story Conversion using LLM

**Description:** This algorithm utilizes an open-source LLM to convert user-specified requirements into detailed user stories, which are then used to guide the code refactoring process.

#### 4.2.2.1 Pseudocode

```

function convert_requirements_to_user_stories(requirements):
    user_stories = []
    for req in requirements:
        story = llm_generate_user_story(req)
        user_stories.append(story)
    return user_stories

function llm_generate_user_story(requirement):
    prompt = f"Convert the following requirement into a user story:\n{requirement}"
    user_story = llm_model.generate(prompt)
    return user_story

```

## 4.2.3 Module 3: Code Refactoring Utilizing DeepSeek-R1-Distill-Qwen-32B

### Algorithm: AI-Driven Code Refactoring

**Description:** This algorithm employs DeepSeek-R1-Distill-Qwen-32B, an advanced LLM, to refactor the Python code based on the generated user stories. It ensures that the refactored code aligns with user requirements while enhancing code quality.

### 4.2.3.1 Pseudocode

```
function refactor_code(original_code, user_stories):
    for story in user_stories:
        refactored_code = llm_refactor(original_code, story)
        original_code = refactored_code
    return original_code

function llm_refactor(code, user_story):
    prompt = f"Refactor the following Python code based on the user story:\n{user_story}"
    refactored_code = llm_model.generate(prompt)
    return refactored_code
```

## 4.3 External APIs/SDKs

RefactoAI integrates several third-party APIs and SDKs to enhance its functionality and streamline the development process. Table [4.1](#) outlines the external APIs/SDKs utilized in the project, their descriptions, purposes, and specific endpoints or functions employed.

API/SDK and Version	Description	Purpose of Usage	API End-point/Function/Class Used
<b>FastAPI (v0.95)</b>	High-performance web framework for building APIs with Python	Develop the backend server, handle HTTP requests, and manage API endpoints	FastAPI, UploadFile, File, Form, HTTPException
<b>Transformers (v4.30)</b>	Library for state-of-the-art Natural Language Processing	Load and utilize LLMs like Phi-4-Mini-Instruct and DeepSeek-R1-Distill-Qwen-32B for user story generation and code refactoring	LlamaForCausalLM, LlamaTokenizer, AutoTokenizer, AutoModelForCausal, CodeLlamaTokenizer
<b>PyTorch (v1.13)</b>	Deep learning framework for model training and inference	Manage model loading, inference, and GPU memory optimization	<code>torch.cuda.</code>
<b>PyCodestyle (v2.9)</b>	Python tool to check your code against PEP8	Perform PEP8 compliance checks on uploaded and refactored Python code	<code>.StyleGuide</code> , <code>.BaseReport</code>
<b>Radon (v5.1)</b>	Python tool for analyzing code complexity and maintainability	Analyze code complexity and maintainability metrics of the uploaded and refactored code	<code>.complexity.cc_visit</code> , <code>.metrics.mi_visit</code> , <code>.metrics.mi_rank</code>
<b>React (v18)</b>	JavaScript library for building user interfaces	Develop the frontend interface for users to upload code, specify requirements, and view results	React, axios
<b>Axios (v1.4)</b>	Promise-based HTTP client for the browser and Node.js	Handle HTTP requests from the frontend to the backend API	<code>axios.post()</code>
<b>Uvicorn (v0.21)</b>	Lightning-fast ASGI server	Serve the FastAPI backend application	<code>uvicorn.run()</code>

Table 4.1: External APIs and SDKs Used in RefactoAI

## 4.4 Testing Details

Ensuring the reliability and efficiency of RefactoAI is paramount. In this iteration, basic testing was conducted to validate the core functionalities of the system. Comprehensive testing, including extensive unit tests and performance benchmarks, will be carried out in the next iteration. The following sections detail the basic testing performed.

### 4.4.1 Integration Testing

Integration testing ensures that different modules of RefactoAI work seamlessly together. For instance, after validating the code in Module 1, the system should accurately pass the validated code to Module 2 for requirement engineering without data loss or corruption.

#### 4.4.1.1 Example of Integration Testing

**Scenario:** Validate that the user requirements input in Module 2 correctly influence the refactoring process in Module 3.

**Test Steps:**

1. Upload a valid Python script through the UI.
2. Input specific requirements for code enhancement.
3. Trigger the refactoring process.
4. Verify that the refactored code reflects the specified requirements.

**Expected Outcome:** The refactored code should incorporate the user-defined requirements, demonstrating the successful integration between requirement engineering and code refactoring modules.

### 4.4.2 System Testing

System testing evaluates the complete RefactoAI system's functionality, performance, and reliability. It ensures that the system meets all specified requirements and operates as intended under various conditions.

#### 4.4.2.1 Example of System Testing

**Test Case:** Assess the system's performance when processing a moderate-sized Python script.

**Test Steps:**

1. Upload a Python script via the web interface.
2. Input specific requirements for code modification.
3. Initiate the refactoring process.
4. Verify that the output code is correctly modified and that the system remains responsive.

**Expected Outcome:** The system should refactor the script within an acceptable time frame and maintain high code quality standards.

#### 4.4.3 React Application Testing

During the testing phase, specific attention was given to the React frontend application to ensure seamless integration with the backend and accurate display of results. The following details outline the testing performed on the React app.

##### 4.4.3.1 Test Input

- **Upload Python File:** `ai.py`
- **Enter Your Requirements:**

`add a bmi function that takes height and weight as input and returns the bmi`

##### 4.4.3.2 Code Quality Analysis

Upon uploading the Python file and submitting the requirements, the system performed a code quality analysis using Radon. The results are as follows:

- `dijkstra` (line 6): Cyclomatic Complexity = 8
- `bellman_ford` (line 34): Cyclomatic Complexity = 10
- `a_star` (line 57): Cyclomatic Complexity = 8
- `reconstruct_path` (line 85): Cyclomatic Complexity = 3
- Maintainability Index: 57.90 (A)



### 4.4.3.3 Generated User Stories

Using Phi-4-Mini-Instruct, the system converted the user's requirements into detailed user stories:

As a user, I want to be able to enter my height in feet and my weight in pounds, and have the program tell me my body mass index (BMI) value, so that I can determine whether or not I am at a healthy weight.

### 4.4.3.4 Modified Code

Based on the user stories, DeepSeek-R1-Distill-Qwen-32B refactored the original Python code to include the requested BMI function. The modified code is presented below:

```
import heapq

# Dijkstra's Algorithm

def dijkstra(graph, start, end):
    priority_queue = [(0, start)] # (cost, node)
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    previous_nodes = {node: None for node in graph}

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_node == end:
            break

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                heapq.heappush(priority_queue, (distance, neighbor))
```

```

    return reconstruct_path(previous_nodes, start, end)

# Bellman-Ford Algorithm

def bellman_ford(graph, start, end):
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    previous_nodes = {node: None for node in graph}

    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight
                    previous_nodes[neighbor] = node

    # Check for negative-weight cycles
    for node in graph:
        for neighbor, weight in graph[node].items():
            if distances[node] + weight < distances[neighbor]:
                raise ValueError("Graph contains a negative-weight cycle")

    return reconstruct_path(previous_nodes, start, end)

# A* Algorithm (uses a simple heuristic, like straight-line distance)

def a_star(graph, start, end, heuristic):
    priority_queue = [(0, start)]
    g_scores = {node: float('infinity') for node in graph}
    g_scores[start] = 0
    f_scores = {node: float('infinity') for node in graph}
    f_scores[start] = heuristic[start]
    previous_nodes = {node: None for node in graph}

    while priority_queue:
        _, current_node = heapq.heappop(priority_queue)

        if current_node == end:
            break

```

```
    for neighbor, weight in graph[current_node].items():
        tentative_g_score = g_scores[current_node] + weight

        if tentative_g_score < g_scores[neighbor]:
            g_scores[neighbor] = tentative_g_score
            f_scores[neighbor] = g_scores[neighbor] + heuristic[neighbor]
            previous_nodes[neighbor] = current_node
            heapq.heappush(priority_queue, (f_scores[neighbor], neighbor))

    return reconstruct_path(previous_nodes, start, end)

# Function to reconstruct the shortest path

def reconstruct_path(previous_nodes, start, end):
    path = []
    current_node = end
    while current_node is not None:
        path.append(current_node)
        current_node = previous_nodes[current_node]
    path.reverse()

    if path[0] == start:
        return path
    else:
        return None # No path found

# Example graph (undirected, weighted)
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1, 'E': 3},
    'E': {'D': 3}
}

# Heuristic for A* (this can be the estimated cost to the end node, simplified here)
heuristic = {
    'A': 7, 'B': 6, 'C': 2, 'D': 1, 'E': 0
}
```

---

```

# Test the algorithms
start_node = 'A'
end_node = 'E'

print("Dijkstra's Algorithm:", dijkstra(graph, start_node, end_node))
print("Bellman-Ford Algorithm:", bellman_ford(graph, start_node, end_node))
print("A* Algorithm:", a_star(graph, start_node, end_node, heuristic))

# Function to calculate the body mass index (BMI)

def bmi_calculator(height, weight):
    # Convert height from feet to meters and weight from pounds to kilograms
    height_meters = height * 0.3048
    weight_kilograms = weight * 0.453592
    bmi = weight_kilograms / (height_meters * height_meters)
    if bmi < 18.5:
        return f"Underweight (BMI: {bmi:.2f})"
    elif bmi < 25:
        return f"Normal weight (BMI: {bmi:.2f})"
    elif bmi < 30:
        return f"Overweight (BMI: {bmi:.2f})"
    elif bmi < 35:
        return f"Moderately obese (BMI: {bmi:.2f})"
    elif bmi < 40:
        return f"Severely obese (BMI: {bmi:.2f})"
    else:
        return f"Very severely obese (BMI: {bmi:.2f})"

# Test the bmi_calculator function
height = float(input("Enter your height in feet: "))
weight = float(input("Enter your weight in pounds: "))
print(f"A person who is {height} feet tall and weighs {weight} pounds has a BMI of: ")

```

#### 4.4.4 Requirement Report Generation

After refactoring and regeneration, the system performs semantic analysis to verify whether each user story has been implemented in the final code. It generates a report indicating which requirements are met and highlights the corresponding lines or functions.

Table 4.2: Requirement Report

User Story ID	User Story	Requirement Met	Matched Elements
REQ1	As a data analyst, I want to be able to quickly calculate summary statistics like mean and median for my datasets, so that I can understand the central tendency and variability of my data.	Met	Function: calculate_stats (Line 71)
REQ2	As an audio engineer, I'd like to have a voice processing tool that can remove noise from my recordings and analyze pitch, allowing me to improve the quality of the audio and understand vocal characteristics.	Met	Line 106: pitch = calculate_pitch(sr, da)
REQ3	Additionally, as someone working with voice data, it would be beneficial for me if I could determine the gender of an individual based on their voice, which could be useful in various applications such as customer service or security systems.	Met	Function: gender_classification (Line 94)

#### 4.4.4.1 Test Results

The React application successfully displayed the modified code with the new BMI function integrated as per the user's requirements. The frontend handled the file upload, requirements input, and result display seamlessly, demonstrating effective interaction between the frontend and backend components.

#### 4.4.4.2 User Experience

Users can now upload their Python scripts and specify their requirements through an intuitive interface. The system provides immediate feedback on code quality and integrates new features as requested, enhancing the overall user experience.

#### 4.4.4.3 Some More Testing Images

To illustrate the testing process and user interface interactions, the following images showcase various frontend scenarios encountered during testing:

- **Frontend Interface: Python File Upload and Analysis**

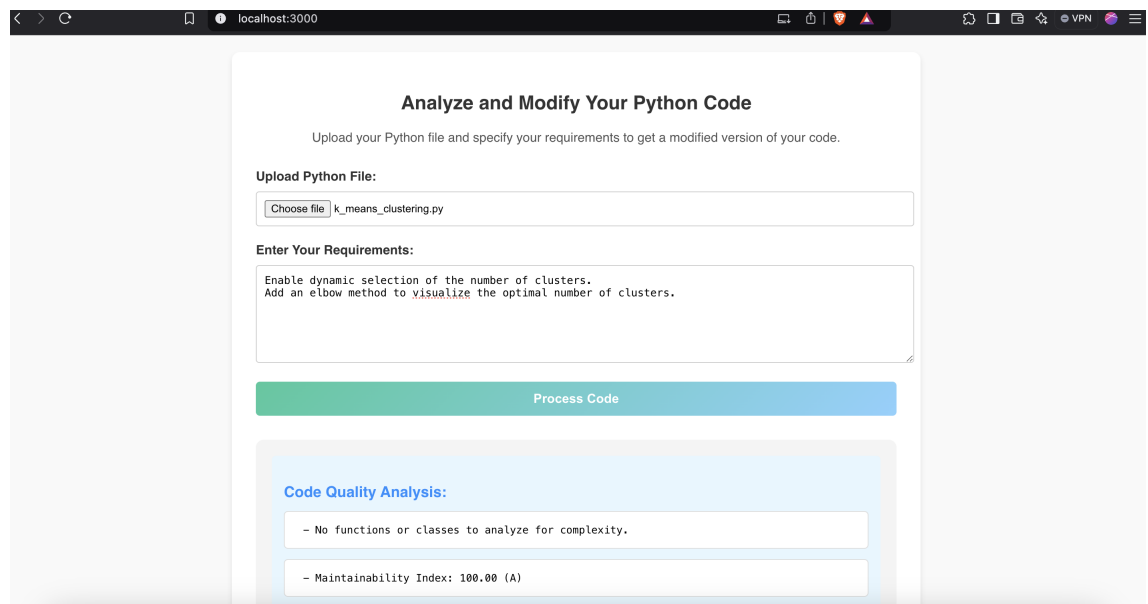


Figure 4.1: Frontend Interface: Python File Upload and Analysis

Displays the front-end interface where users upload a Python file for analysis, showing the file selector, upload progress, and initial parsing feedback.

- **Frontend Interface: User Stories Generation**

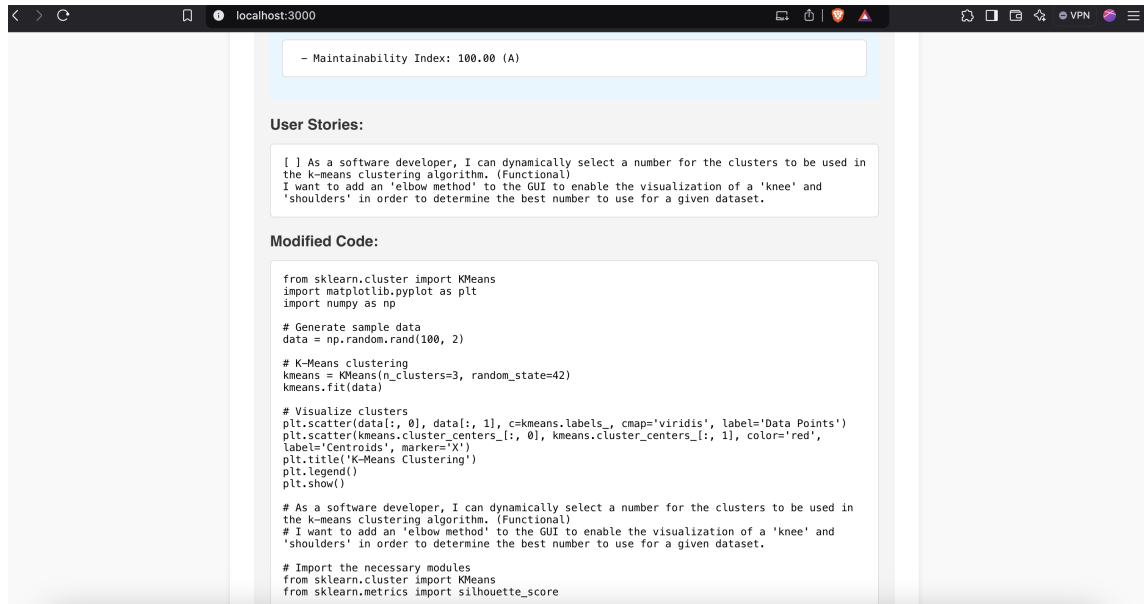


Figure 4.2: Frontend Interface: User Stories Generation

Illustrates the view in which the application generates and displays user stories derived from the uploaded code, including mapping to requirements.

- **Frontend Interface: Displaying Results and Modified Code**

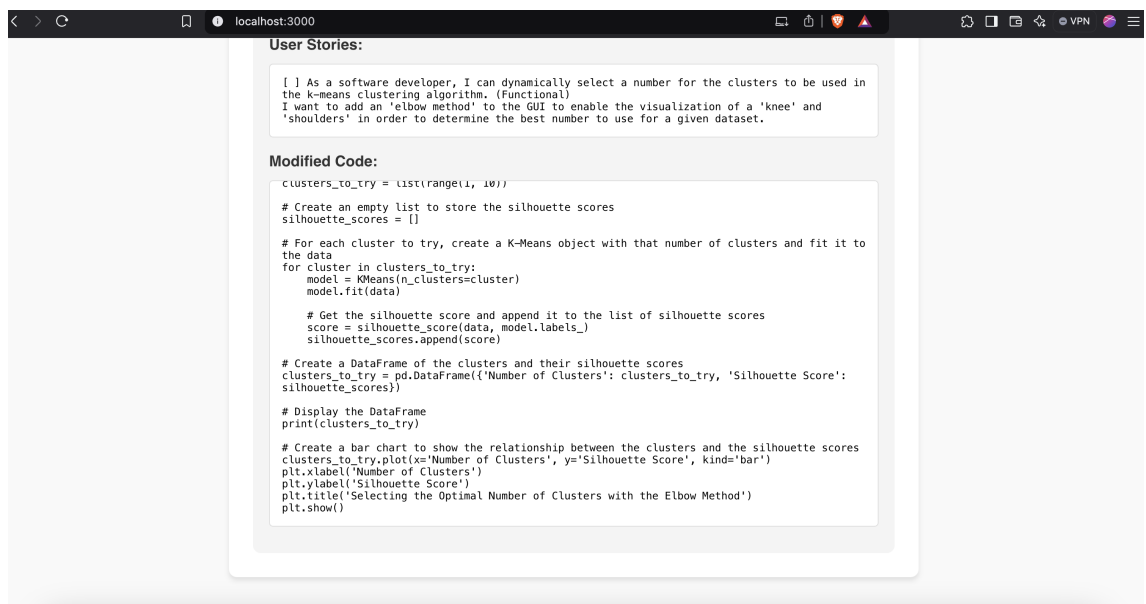
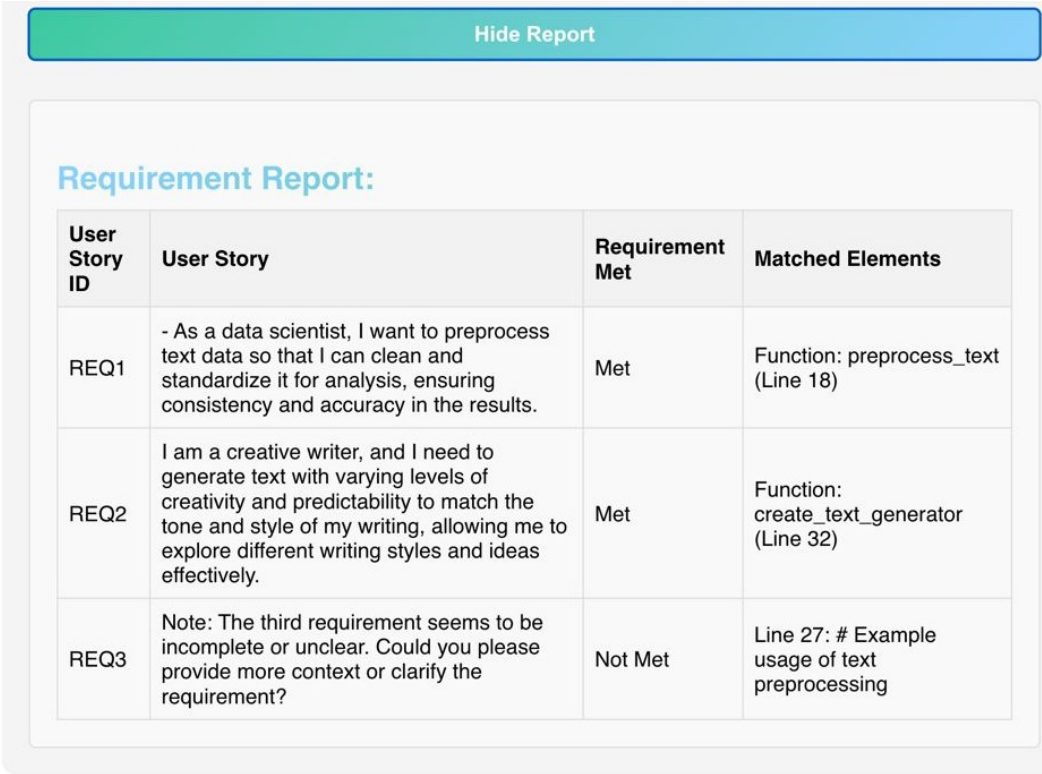


Figure 4.3: Frontend Interface: Displaying Results and Modified Code

Shows the final front-end screen presenting the analysis results alongside the modified code, highlighting changes made to meet specified requirements.

- **Requirement Report**



User Story ID	User Story	Requirement Met	Matched Elements
REQ1	- As a data scientist, I want to preprocess text data so that I can clean and standardize it for analysis, ensuring consistency and accuracy in the results.	Met	Function: preprocess_text (Line 18)
REQ2	I am a creative writer, and I need to generate text with varying levels of creativity and predictability to match the tone and style of my writing, allowing me to explore different writing styles and ideas effectively.	Met	Function: create_text_generator (Line 32)
REQ3	Note: The third requirement seems to be incomplete or unclear. Could you please provide more context or clarify the requirement?	Not Met	Line 27: # Example usage of text preprocessing

Figure 4.4: Requirement Report

The detailed requirement report generated by the application, outlining which user stories have been met and associating each with the specific function names or line numbers in the processed code.

- **Libraries-Only File**



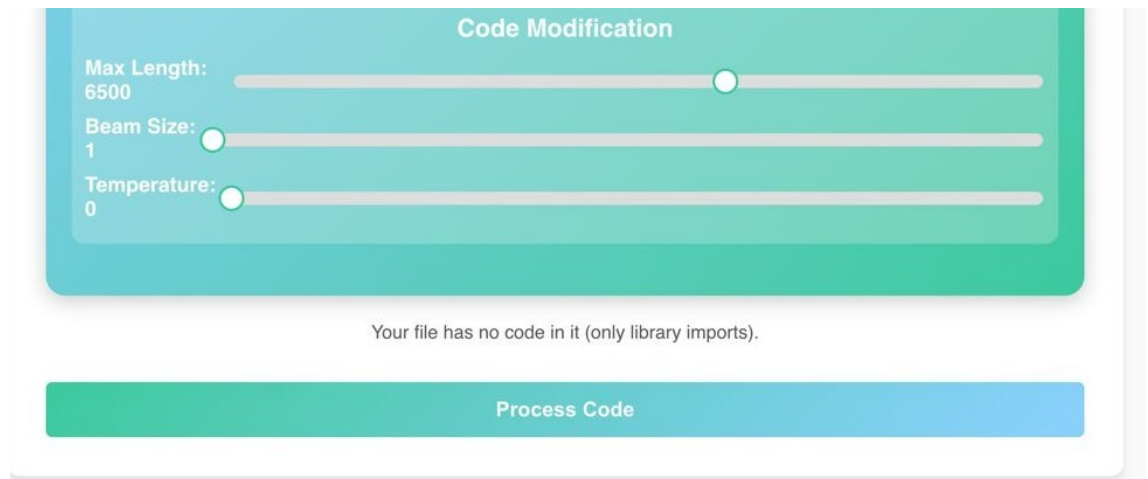


Figure 4.5: Libraries-Only File

Indicates that the uploaded Python file contains no executable code—only library import statements—so there is nothing to process beyond the imports.

- **Empty File Uploaded**

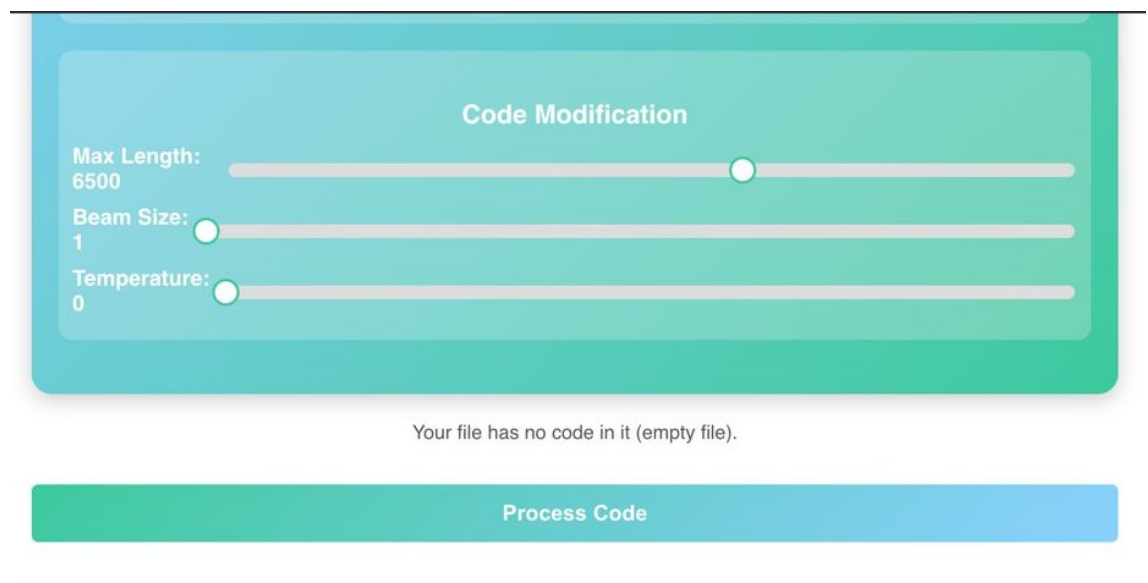


Figure 4.6: Empty File Uploaded

Illustrates the scenario where a completely empty file is uploaded, prompting the application to display a message that no code is available for processing.

- **Syntax Error Detected**

The screenshot shows a 'Code Modification' interface with three sliders: 'Max Length' (6500), 'Beam Size' (1), and 'Temperature' (0). Below the sliders, a message states: 'Syntax errors detected: SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='? at line 74, column 12'. At the bottom is a 'Process Code' button.

Figure 4.7: Syntax Error Detected

Demonstrates the system detecting a syntax error in the uploaded code, providing detailed information about the nature of the error and its exact location (line and column).


- **Missing Requirements Prompt**

The screenshot shows a 'Code Modification' interface with three sliders: 'Max Length' (6500), 'Beam Size' (1), and 'Temperature' (0). Above the sliders, a message states: 'Please enter your requirements.'. At the bottom is a 'Process Code' button.

Figure 4.8: Missing Requirements Prompt

Shows the application's response when the user attempts to process code without entering any requirements—prompting the user to supply the necessary requirement descriptions.

- **Modified Code Output**



The screenshot shows a code editor with a title bar "Modified Code:". The code is a Python script for building and training a neural network model. The code is divided into three sections, each with a light green background highlight. The first section defines the model architecture with layers: Conv2D, MaxPooling2D, Flatten, Dense, and Softmax. The second section defines the build\_nn\_model function. The third section defines the train\_model function. A "Hide Report" button is visible at the bottom of the code block.

```

layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
layers.MaxPooling2D((2, 2)),
layers.Conv2D(64, (3, 3), activation='relu'),
layers.MaxPooling2D((2, 2)),
layers.Conv2D(128, (3, 3), activation='relu'),
layers.MaxPooling2D((2, 2)),
layers.Flatten(),
layers.Dense(128, activation='relu'),
layers.Dense(10, activation='softmax')
})
return model
# Define Neural Network model
def build_nn_model():
    model = models.Sequential([
        layers.Flatten(input_shape=(32, 32, 3)),
        layers.Dense(128, activation='relu'),
        layers.Dense(128, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    return model
# Compile and train model
def train_model(model, name):
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
['accuracy'])
    history = model.fit(data_augmentation.flow(x_train, y_train, batch_size=32),

```

Figure 4.9: Modified Code Output

Presents the modified code generated by the system in response to the specified requirements, clearly highlighting the added or altered sections that fulfill the user's requests.

## 4.5 Current Iteration Summary

As of the current iteration, the following enhancements have been implemented and tested:

- **Backend Development** (`backend/main.py`)
  - **Semantic Validation Module:** Integrated a transformer-based semantic similarity component to automatically assess whether each user requirement is reflected in the code, using a dynamic threshold.
  - **Dynamic Threshold Logic:** Implemented logic to adjust the minimum similarity score required for a requirement to be considered satisfied, improving the robustness of requirement coverage checks.
- **Frontend Development** (`src/App.js`)
  - **Hyperparameter Controls:** Added interactive sliders and input fields for users to modify LLM generation hyperparameters (e.g., `temperature`, `max_tokens`, `top_p`) directly in the web interface.
  - **Real-Time Similarity Feedback:** Enhanced the UI to display per-requirement similarity scores and pass/fail status in the generated report, giving immediate insight into which requirements were met.
- **Testing Enhancements**
  - **Report Testing:** Developed an automated test suite for the semantic similarity report, verifying correct classification of requirements at multiple threshold levels.
  - **Hyperparameter Validation:** Created test cases to ensure that user-adjusted hyperparameters influence LLM outputs as expected while maintaining system stability.

## 4.6 Conclusion

The latest iteration of RefactoAI has significantly advanced the system's capability to validate and report on requirement satisfaction through transformer-based semantic similarity checks with a dynamic threshold. By exposing LLM hyperparameters in the frontend, users now have fine-grained control over generation behavior, enhancing flexibility and customization. The expanded automated test suites ensure both the accuracy of semantic

coverage assessments and the reliability of hyperparameter adjustments. Moving forward, future work will focus on further refining the semantic scoring algorithms, extending support for additional LLM architectures, and conducting performance optimizations to handle larger codebases while preserving interactive responsiveness. These improvements continue to drive RefactoAI toward a fully automated, intelligent code refactoring solution that delivers high quality and developer productivity gains.

# Bibliography

- [1] ACM. Acm transactions on software engineering and methodology. *ACM Transactions on Software Engineering and Methodology*, 2024.
- [2] Capgemini. Automated testing efficiency. *Capgemini Insights*, 2024.
- [3] McKinsey & Company. It project budgets and code quality. *McKinsey Digital Insights*, 2024.
- [4] Ralph E. Johnson and Michele S. N. B. Refactoring-induced bugs. *Proceedings of the 12th International Conference on Software Maintenance*, 2012.
- [5] Miryung Kim. Developer time spent on refactoring. *IEEE Transactions on Software Engineering*, 2014.
- [6] OpenAI. Performance optimization. *XallyAI*, 2024.