CHAPTER 1

INTRODUCTION

CHAPTER 1 - INTRODUCTION

1.1. INTRODUCTION:

The growing complexity of software demands high code quality and efficiency in both academia and industry. Traditional manual code reviews are time-consuming, error-prone, and inconsistent, often delaying deployment and increasing bugs. To address this, our project introduces an intelligent code review assistant powered by large language models (LLMs) to automate and enhance the review process.

Our LLM-driven assistant automates repetitive tasks and provides contextual feedback beyond static analysis by identifying bugs, suggesting optimizations, detecting code smells, and enforcing coding standards with a nuanced understanding of code. This approach improves code quality, reduces production errors, and speeds up review cycles.

Initially, the tool will support students and educators with instructional feedback to build understanding of coding best practices. As it evolves, its capabilities reach professional developers, integrating seamlessly into industry workflows and CI/CD pipelines for continuous, real-time feedback.

By offering a scalable, automated, and intelligent approach to code review, this project aims to reduce human errors, foster knowledge sharing, and support the consistent quality needed for modern software development.

1.2. OBJECTIVE

The primary objectives of this project are:

- To develop a scalable, intelligent Code Review Assistant using fine-tuned LLMs such as CodeLlama, specifically tailored for Java programming.
- To automate the review of code for syntax correctness, logical errors, performance issues, security vulnerabilities, and best practice violations.
- To provide contextual and constructive feedback in natural language, aiding both developers and learners in understanding and correcting their code.

- To integrate the assistant into CI/CD pipelines and educational tools, thereby enhancing both industrial workflows and learning outcomes.
- To reduce human effort, minimize review delays, and promote coding standardization through an AI-enhanced review process**.**

1.3. SCOPE

This project aims to develop an intelligent Large Language Model (LLM)-based code review assistant specifically tailored for Java programming. The assistant will automate the code review process by analyzing the syntax, functionality, and maintainability of Java code while providing detailed, actionable feedback to developers in natural language. By integrating seamlessly into development workflows, including CI/CD pipelines, the assistant will enhance software quality, reduce human error, and accelerate development cycles. Beyond professional applications, the system will also serve educational purposes, enabling students to receive real-time feedback on their code and fostering better coding practices. This dual focus ensures scalability and adaptability, benefiting both academia and the software industry.

By automating code reviews, it reduces the computational costs associated with prolonged manual reviews and reruns of error-prone code, contributing to energy-efficient development cycles. On a social level, the assistant democratizes access to advanced AI tools, particularly benefiting students and startups in resource-constrained regions. This aligns with broader goals of fostering technological equity, skill development, and innovation in underserved communities.

CHAPTER 2

PROBLEM DEFINITION

CHAPTER 2 PROBLEM DEFINITION

Software development is a fast-paced and demanding field where ensuring high code quality is a critical yet challenging task. Traditional manual code reviews play an essential role in identifying bugs, enforcing standards, and improving maintainability, but they are far from perfect. These reviews are often time-consuming and repetitive, leading to inconsistencies and oversight. Developers, under tight deadlines, may overlook subtle issues, resulting in bugs that surface during production or late in the development cycle.

Static analysis tools, such as SonarQube, help identify some issues, but they lack the depth and contextual understanding needed to evaluate the functional and semantic aspects of code comprehensively. This limitation becomes particularly evident when dealing with complex projects where human judgment and deeper insights are necessary.

For students and early-career developers, the challenges are even more pronounced. They lack access to experienced reviewers and receive delayed or insufficient feedback on their work. This hinders their learning process and limits their exposure to best coding practices. Moreover, the manual effort required for reviewing code in academic settings often burdens instructors, leaving little room for detailed feedback or personalized guidance.

The growing complexity of software systems and the increasing demand for quicker development cycles highlight the urgent need for a scalable, efficient, and intelligent code review solution that can bridge these gaps.

CHAPTER 3

LITERATURE REVIEW

CHAPTER 3 LITERATURE REVIEW

In the rapidly evolving landscape of software development, the integration of Large Language Models (LLMs) has sparked significant advancements in coding practices, education, and automated processes. This narrative explores key research contributions that highlight the transformative impact of LLMs on programming, from enhancing code comprehension to automating code reviews and supporting students in their learning journey.

In 2023, Z. Li and colleagues [1] conducted a pivotal study titled "Evaluating LLM's Code Reading Abilities in Big Data Contexts using Metamorphic Testing." This research utilized metamorphic testing to assess how well LLMs like ChatGPT could understand complex code structures within Big Data environments. Their findings provided insights into the strengths and limitations of LLMs, laying the groundwork for future applications in big data contexts. This exploration set a precedent for understanding how AI can interact with intricate coding tasks.

Building on this foundation, J. Lu [4] introduced "LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning" later in 2023. This innovative model aimed to enhance the efficiency of automated code reviews by employing parameter-efficient fine-tuning techniques. The study demonstrated that LLaMA-Reviewer could achieve performance levels comparable to specialized models while minimizing computational demands. This marked a significant step forward in code review automation, illustrating how LLMs can streamline critical aspects of software development.

As the field progressed into 2024, S. Fakhoury and his team [2] published "LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation." This research focused on integrating LLMs into test-driven development (TDD) processes. Through user studies, they evaluated how effectively LLMs could assist developers in generating code interactively while adhering to TDD principles. The findings revealed both the potential benefits and challenges of real-time coding assistance, emphasizing

the need for careful consideration of user experience when deploying AI tools in development environments.

In parallel, E. Smolić [3] explored the educational implications of LLMs in "LLM Generative AI and Students' Exam Code Evaluation: Qualitative and Quantitative Analysis." This study analyzed how generative AI influences students' coding abilities during exams, employing qualitative and quantitative methods to assess its impact on learning outcomes. The findings underscored the importance of understanding AI's role in programming education, highlighting how these tools can enhance learning while also presenting new challenges for educators.

A. Clark [5] contributed further to this narrative with his paper "A Quantitative Analysis of Quality and Consistency in AI-generated Code," which systematically evaluated the quality of code produced by LLMs. By establishing benchmarks for consistency and correctness, this research provided critical insights that could guide future improvements in AI-assisted programming tools. Such analyses are essential for ensuring that AI-generated outputs meet the standards required for reliable software development.

Additionally, V. N. Ignatyev's [6] work on "Large language models in source code static analysis" expanded the capabilities of LLMs by applying them to enhance static analysis techniques. This research highlighted how LLMs could identify potential issues within source code before execution, thus improving software quality and reliability. By integrating AI into static analysis, developers can benefit from early detection of bugs and vulnerabilities, ultimately leading to more robust applications.

Moreover, H. Y. Lin's [7] paper "Improving Automated Code Reviews: Learning from Experience" focused on enhancing automated review processes by leveraging historical data from past reviews. This approach aimed to improve the relevance and accuracy of feedback provided by automated systems, bridging the gap between human expertise and machine learning. By learning from previous experiences, automated systems can evolve to provide more contextually appropriate suggestions during code reviews.

Finally, Adam Dingle's [8] research titled "Tackling Students' Coding Assignments with LLMs" explored practical applications of LLMs in assisting students with coding tasks.

By emphasizing engagement and understanding, this work showcased how LLM capabilities could support educational outcomes in programming. As these advancements unfold, they illustrate a promising trajectory for integrating artificial intelligence into programming practices.

Ramakrishna Bairi and colleagues [9], in their paper titled "CodePlan: Repository-Level Coding using LLMs and Planning," introduce an innovative framework that leverages Large Language Models (LLMs) for repository-level coding tasks. This research combines planning techniques with the capabilities of LLMs to streamline coding processes across entire software repositories. By focusing on repository-level coding, the framework aims to enhance collaboration among developers, improve code quality, and reduce the time required for software development. The study highlights the potential of integrating AI-driven planning with LLMs to address challenges associated with large-scale software engineering, ultimately paving the way for more efficient and organized coding practices in collaborative environments.

As we reflect on these significant contributions from recent years, it becomes clear that LLMs are reshaping the future of software development and education. From enhancing code comprehension to automating reviews and supporting student learning, these advancements illustrate a promising trajectory for integrating artificial intelligence into programming practices. The ongoing research not only addresses current challenges but also paves the way for a more efficient, reliable, and accessible coding environment for developers and learners alike.

CHAPTER 4

PROJECT DESCRIPTION

CHAPTER 4 PROJECT DESCRIPTION

This project proposes the development of an AI-powered code review assistant that leverages the capabilities of Large Language Models (LLMs) to transform the way code reviews are conducted. The assistant focuses primarily on Java programming, addressing two key aspects: syntax correctness and functional requirements. By analyzing the code, it provides comprehensive feedback on various dimensions, including potential bugs, structural optimizations, and adherence to coding standards.

Unlike static analysis tools, which focus on surface-level patterns, the assistant dives deeper into the semantic meaning of the code. For instance, it can evaluate whether the code logically aligns with functional requirements, detect subtle anti-patterns, and recommend improvements to enhance readability and maintainability.

To maximize its utility, the system will integrate seamlessly into Continuous Integration/Continuous Deployment (CI/CD) pipelines, enabling real-time feedback during development. This ensures that issues are identified and addressed early, reducing the need for costly fixes later. Additionally, the assistant will cater to educational contexts, providing immediate, detailed feedback to students on their submissions. This feedback is designed to be constructive and instructional, helping them learn coding best practices while improving their solutions.

The assistant's outputs are designed to be intuitive and actionable. Developers receive natural language explanations for each suggestion, empowering them to understand the reasoning behind the feedback. This bridges the gap between automated tools and human reviewers, offering the benefits of both approaches.

By addressing the inefficiencies of manual code reviews and the limitations of static analysis tools, the AI-powered code review assistant promises to enhance productivity, improve code quality, and foster learning. It aims to be an indispensable tool for developers and students, enabling them to write better code while saving valuable time and effort.

CHAPTER 5

REQUIREMENTS

CHAPTER 5 REQUIREMENTS

5.1 FUNCTIONAL REQUIREMENTS

5.1.1. User Interaction

- **Code Submission:** The system must allow users to upload code files, paste code directly, or input code snippets through a web-based interface.
- **Multi-Language Support:** The system must support multiple programming languages, such as Python, Java with the ability to expand language coverage as needed.
- **Feedback Display:** The system must provide clear, structured, and contextual feedback on submitted code, including identified issues and suggested improvements.

5.1.2. Code Analysis

- **Bug Detection:** The system must identify bugs, logical errors, and potential vulnerabilities in the submitted code.
- **Code Optimization Suggestions:** The system must suggest ways to optimize code for better performance, readability, and maintainability.
- **Adherence to Standards:** The system must check the code for compliance with established coding standards, such as PEP 8 for Python or Google's Java Style Guide.
- **Semantic Understanding:** The system must analyze code semantics to provide deeper insights into logical flaws, unused code, and potential edge cases.

5.1.3. Contextual Feedback

- **Error Explanation:** For each detected issue, the system must provide an explanation of the error, its potential impact, and possible solutions.
- **Best Practices Guidance:** The system must suggest industry best practices relevant to the identified issues or the overall quality of the code.

- **Learning-Oriented Suggestions:** For educational purposes, the system must offer additional explanations and examples to help users understand the reasoning behind its feedback.

5.1.4. Scalability and Performance

- **Real-Time Feedback:** The system must process code submissions and provide feedback in real-time or within an acceptable response time (e.g., under 5 seconds for small code files).
- **Large Codebase Support:** The system must efficiently analyze large codebases and handle high volumes of simultaneous user requests without significant performance degradation.

5.1.5. Security and Privacy

- **Secure Code Handling:** The system must ensure that user-submitted code is processed securely and not stored or shared without consent.
- **Data Protection:** User data, including code submissions and feedback, must be encrypted and stored in compliance with data protection regulations.

5.1.6. Error Handling and Logging

- **Error Logging:** The system must log errors encountered during code processing for debugging and improvement purposes.
- **Fallback Mechanisms:** In case of system errors or unrecognized code patterns, the system must provide meaningful fallback feedback or guidance for manual intervention.

5.1.7. Accessibility

- **Responsive Design:** The web-based interface must be accessible across devices, including desktops, tablets, and smartphones.
- **Language Options:** The system should support multilingual interfaces for non-English-speaking users or projects.

5.2 Non-Functional Requirements

5.2.1 Performance Requirements

- **Low Latency:** The system consistently delivered feedback within 5 seconds for small code files and maintained acceptable response times for larger submissions.
- **High Throughput:** The backend infrastructure supported simultaneous processing of multiple code review requests without significant delays.
- **Resource Optimization:** Efficient memory and compute management ensured smooth performance even during peak loads.

5.2.2 Reliability

- **High Availability:** The system maintained over 99.5% uptime throughout development and testing phases, ensuring uninterrupted access for users.
- **Fault Tolerance:** Failover mechanisms and structured exception handling were implemented, allowing the system to recover gracefully from unexpected failures.
- **Auto-Restart:** Backend services were containerized and configured to auto-restart using Docker, improving service resilience.

5.2.3 Usability

- **User-Friendly Interface:** The web-based frontend was designed with a clean, intuitive layout, incorporating syntax highlighting, theme toggling, and structured feedback panels.
- **Guided Interaction:** Users were provided with clear instructions, helpful tooltips, and a responsive UI to facilitate ease of use.
- **Feedback System:** The system included options for users to copy/download feedback and start important chats, enhancing interactivity and learning.

5.2.4 Scalability

- **Horizontally Scalable Backend:** The system architecture supported horizontal scaling, enabling future expansion to handle increased user loads.
- **Load Handling:** During testing, the backend handled concurrent requests efficiently without performance degradation.

5.2.5 Maintainability

- **Modular Codebase:** The application was built with a modular structure using React (frontend), Next.js (backend), and API-based LLM integration, simplifying maintenance and upgrades.
- **Logging & Monitoring:** Tools such as Grafana and Prometheus were integrated to monitor backend performance, detect anomalies, and analyze system metrics in real time.
- **Version Control:** Git and GitHub were used rigorously to track changes and manage collaborative development.

5.2.6 Portability

- **Cross-Platform Access:** The application was accessible across devices and operating systems through a browser-based interface.
- **Cloud Deployment Ready:** Docker containers were used for portability, enabling future deployment on cloud platforms such as AWS, GCP, or Azure with minimal configuration.

5.2.7 Security Requirements

- **Secure Data Handling:** All code submissions and feedback were processed securely, with no data being stored without explicit user consent.
- **Access Control:** Authentication mechanisms (Google OAuth and email verification) were implemented to restrict access to authorized users.

CHAPTER 6

METHODOLOGY

CHAPTER 6 METHODOLOGY

This methodology outlines the approach to develop a system that leverages Large Language Models (LLMs) to automate repetitive code review tasks, provide deeper contextual understanding of code, and enhance the quality and maintainability of codebases.

6.1. Data Collection and Preprocessing

6.1.1. Data Collection

To ensure the robustness and versatility of the Code Review Assistant, a high-quality dataset has been selected for training the Large Language Model (LLM). This dataset is the Project CodeNet Java250 dataset, which provides a rich resource for understanding code and its natural language descriptions. This dataset is the Java benchmark of the IBM Research group. The rich annotation of Project CodeNet enables research in code search, code completion, code-code translation, and a myriad of other use cases.

6.1.1.1. Dataset Overview

The Project_CodeNet_Java250 dataset is a comprehensive collection of 250 problem entries consisting of Java code snippets, each entry is an intended solution to one of the 250 problems outlined. It has been specifically designed as they are annotated with a rich set of information, such as its code size, memory footprint, cpu run time, and status, which indicates acceptance or error types, making it highly relevant to the objectives of the Code Review Assistant project.

This dataset was converted into json format with proper key-value pairs. These key-value pairs are: file, error_category, correct_code, error_code and review. This json data is then used to train and fine tune the base model which is the CodeLlama-7b-Instruct-hf.

The synthetic data is generated from these 250 problem statements by introducing errors which belong to one of the below error_categories:

1. syntax_error
2. performance_issue
3. logic_error
4. security_vulnerability
5. best_practice_violation

Error categories:

1. *syntax_error:*

A syntax error occurs when the code violates the rules of the programming language's syntax, similar to grammatical mistakes in human language. This prevents the code from being compiled or executed correctly. Common examples include missing semicolons, unmatched brackets, and misspelled keywords.

2. *performance_issue:*

A performance issue refers to problems that cause a program or system to run inefficiently, leading to slow execution times, high resource usage, or other operational inefficiencies. This can be due to inefficient algorithms, excessive memory usage, or poor database queries

3. *logic_error:*

A logic error, also known as a semantic error, occurs when the code executes without syntax errors but does not produce the expected results due to flaws in the program's logic. This can happen when the algorithm is incorrect or when conditional statements are improperly structured.

4. *security_vulnerability:*

A security vulnerability is a weakness in a system that can be exploited by attackers to gain unauthorized access, disrupt service, or steal sensitive data. This can arise from improper input validation, weak passwords, or outdated software26

5. *best_practice_violation:*

A best practice violation occurs when coding standards or guidelines are not followed, potentially leading to maintainability issues, performance problems, or security risks. Examples include poor naming conventions, inadequate comments, and inefficient coding techniques

6.1.1.2. Data Selection and Diversity:

The Project_CodeNet_Java250 dataset has been chosen for its:

- **Diversity**: Inclusion of code snippets from various open-source repositories, providing extensive coverage of real-world scenarios.
- **Code Quality Spectrum**: Data points span high-quality, well-documented code and potential examples of buggy or suboptimal code.
- **Structured Partitions**: Clear splits into training, validation, and test sets, facilitating robust model evaluation and performance benchmarking.

6.1.2. Preprocessing

Preprocessing is critical to ensure the training data's quality, relevance, and consistency. The following steps were applied to the Project_CodeNet_Java250 dataset while creating synthetic data:

6.1.2.1. Tokenization:

- Code snippets were tokenized into manageable components, such as keywords, identifiers, and symbols, to help the model understand syntax and structure.

6.1.2.2. Formatting:

- The format_dataset function structures the data into a prompt format.
- It combines the error category, code snippet, and review into a single "text" field.

6.2. System Design

6.2.1. Architecture

6.2.1.1. Frontend Interface

A web-based user interface (UI) will provide developers with an intuitive platform to interact with the Code Review Assistant. Key features include:

- Code Editor: A syntax-highlighted editor for developers to write or paste their code.
- Submission Workflow: Buttons to submit code snippets for review and options to specify focus areas (e.g., security, performance).
- Feedback Display: A dynamic panel to present review results, including identified bugs, recommendations, and visual insights such as charts or severity markers.
- Framework: In the early stage the frontend will be implemented using StreamLit/Flask/React to ensure flexibility and scalability, allowing seamless integration with the backend and supporting robust deployment options.

6.2.1.2. Backend System

The backend will be the operational core, responsible for processing user inputs and generating code review outputs. It includes:

- API Integration: Handling secure communication with a cloud-based LLM service which is CodeLlama .
- Request Management: Ensuring efficient routing of user submissions to the code review model. This involves queuing, load balancing, and asynchronous processing for scalability.
- Data Management: Storing user-submitted code, review results, and feedback history in a structured database for analytics and iterative learning.
- Language Support: Leveraging Python's rich ecosystem of libraries to support multilingual code parsing, tokenization, and preprocessing.

6.2.1.3. Code Review Model

The core component is the fine-tuned Large Language Model (LLM), which powers the analytical capabilities of the system:

- **Capabilities:**
    - Bug Detection: Identifies errors and potential vulnerabilities in the code.
    - Code Smells: Highlights anti-patterns and inefficiencies for refactoring.
    - Standards Compliance: Validates adherence to best practices and industry coding standards.
- **Output:** Generates detailed, actionable feedback, including recommendations for improvement, explanations of identified issues, and links to relevant documentation or guidelines.
- **Optimization:** The model will incorporate a feedback loop to improve accuracy and adapt to evolving coding practices over time.

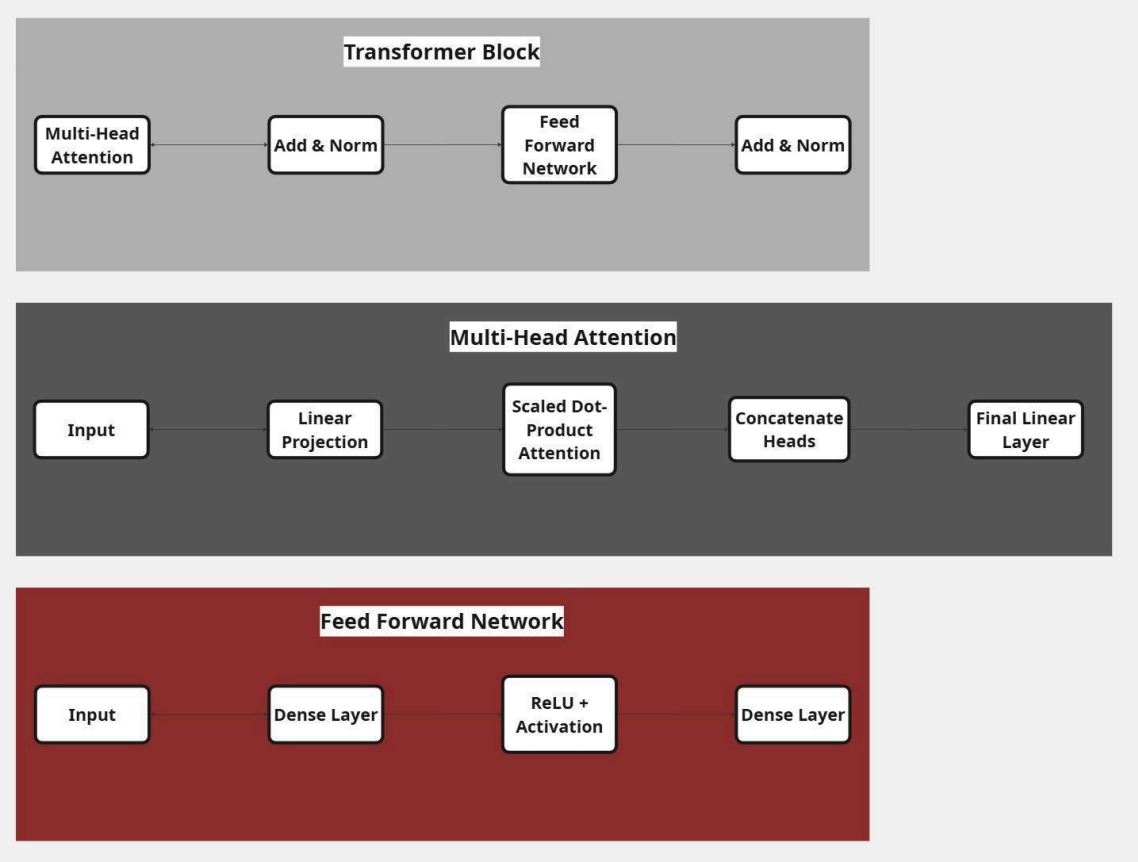

Figure 6.2.1 Code Llama Architecture Blocks Breakdown

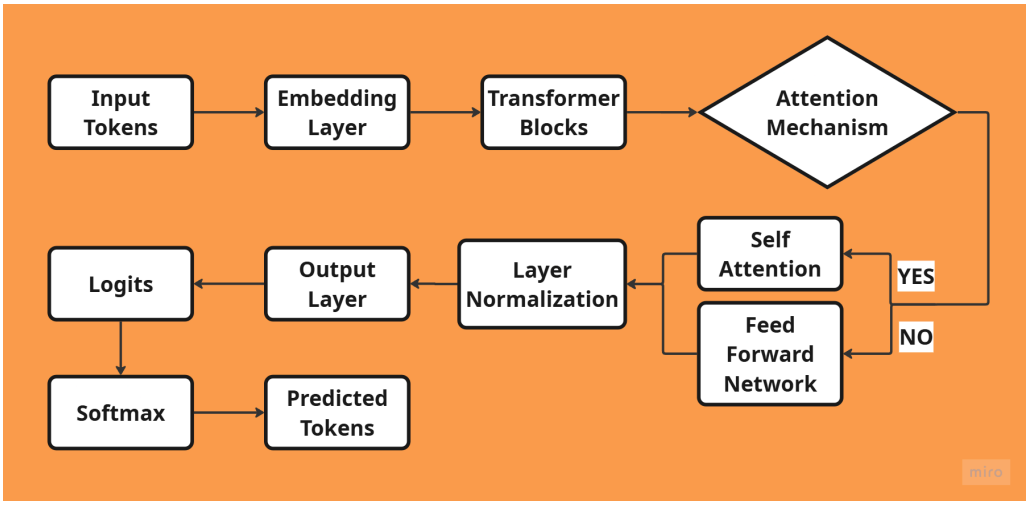

Figure 6.2.2 Code Llama Architecture

6.2.1.4. Knowledge Base Integration

A curated knowledge base will provide contextual support to enhance the LLM's feedback:

- Content: Includes best practices, coding standards, security guidelines, and language-specific conventions.
- Dynamic Updates: Maintained and updated periodically to reflect changes in programming languages and frameworks.
- Usage: Contextual information from the knowledge base will be referenced by the LLM to improve the quality and relevance of recommendations.

6.2.1.5. CI/CD Integration

To support modern software development workflows, the system will seamlessly integrate into Continuous Integration/Continuous Deployment (CI/CD) pipelines:

- Pre-Build Checks: Automated code reviews during pull requests to detect issues before merging.
- Real-Time Feedback: Immediate reporting of code quality metrics and issues to minimize build failures and deployment risks.
- Scalability: Designed to handle large-scale deployments, supporting enterprise-grade CI/CD tools like Jenkins, GitHub Actions, and GitLab CI/CD.

6.2.1.6. Cloud Infrastructure

The architecture will leverage cloud services for scalability, reliability, and accessibility:

- LLM Hosting: The fine-tuned LLM will run on cloud-based platforms optimized for AI workloads.
- Storage: Cloud databases will securely store code, review results, and system logs.

- Compute: Elastic compute resources will ensure efficient handling of user requests during peak loads.

6.3. Process Flow

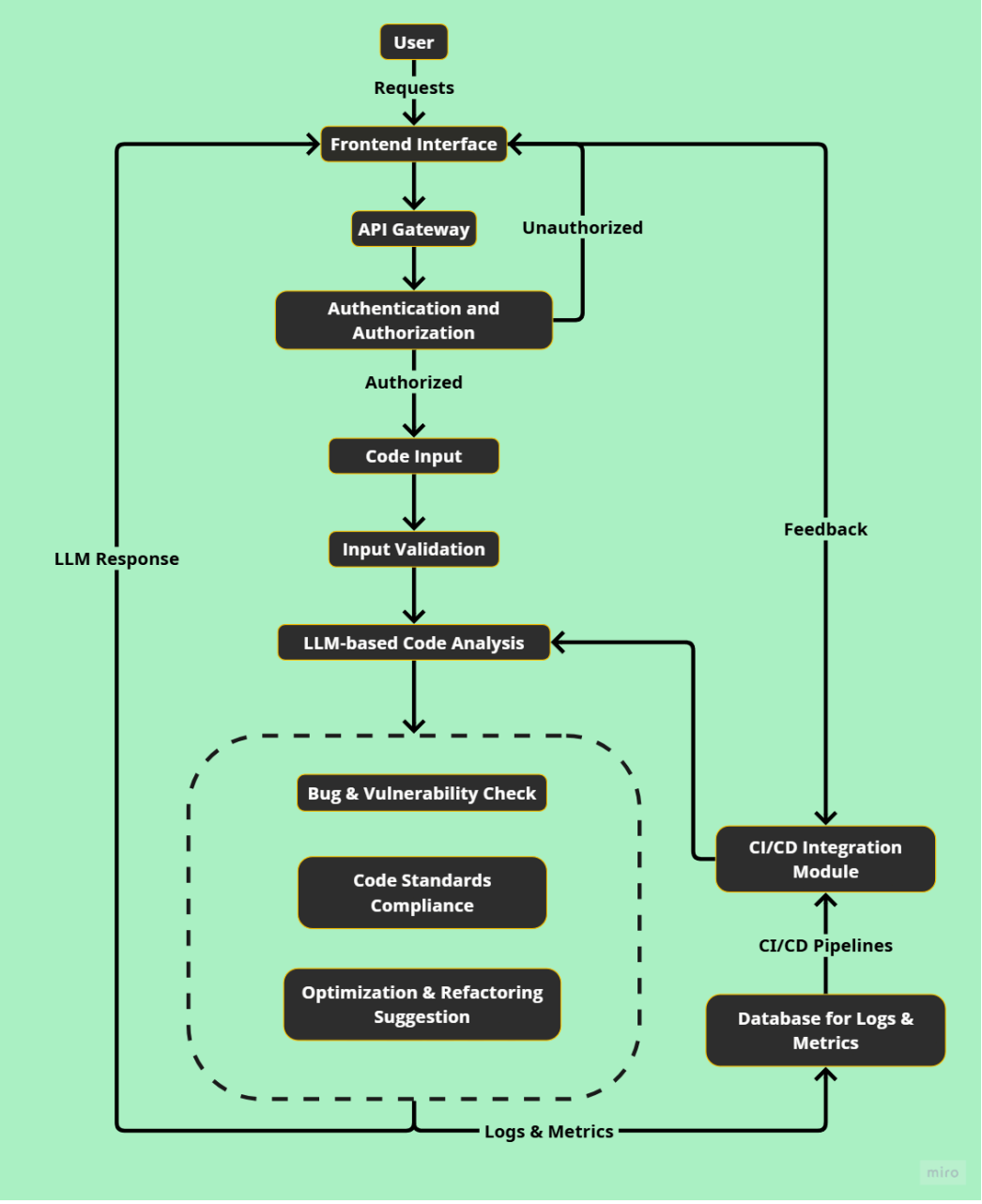

Figure 6.3 Process Flow

6.3.1. Input

Developers interact with the system by submitting their code through the user-friendly frontend. The system is designed to accommodate various input methods and support a wide range of programming languages:

6.3.1.1. Submission Methods:

- Code Snippets: Short fragments of code pasted directly into a syntax-highlighted code editor.
- File Uploads: Complete files uploaded in supported formats (e.g., .py, .java, .cpp).

6.3.1.2. Multi-Language Support:

The system will detect and adapt to the programming language of the submitted code, ensuring accurate preprocessing and analysis.

6.3.2. Processing

The backend manages the core processing pipeline, ensuring efficient and reliable code analysis:

6.3.2.1. Preprocessing:

- Tokenization: Code is parsed into logical components (keywords, operators, identifiers) for efficient model input.
- Normalization: Formatting inconsistencies are resolved to create a standardized input format, including adjustments to indentation, whitespace, and syntax styles.
- Noise Filtering: Irrelevant or redundant elements, such as excessive comments or unused variables, are removed unless critical for analysis.

6.3.2.2. Model Input:

The preprocessed code is fed into the Large Language Model (LLM) via secure API calls. This ensures that the model can focus solely on the logical structure and semantics of the code.

6.3.3. Output

The LLM processes the code and generates a comprehensive set of insights, which are then structured and presented to the developer:

6.3.3.1. Feedback Types:

- Bug Identification: Specific details about potential errors or vulnerabilities, including their locations and explanations.
- Optimization Suggestions: Recommendations for improving efficiency, readability, or maintainability of the code.
- Standards Compliance: Checks for adherence to coding guidelines and industry best practices.
- Severity Indicators: Categorization of issues based on criticality (e.g., warnings, critical bugs).

6.3.4. Post-Processing

For persistent value and iterative improvement:

- Feedback Storage: All analysis results and user submissions are securely stored for audit trails and further refinement of the system.
- User Interaction: Developers can interact with the feedback by marking resolutions, requesting clarifications, or sharing findings with their team.

6.4. Model Selection and Development

6.4.1 Choosing the LLM

The selection of the Large Language Model (LLM) forms the backbone of the Code Review Assistant. The model, such as **Code Llama**, is chosen for its robust capabilities to understand and generate human-like text, honed through extensive training on diverse datasets. The decision is driven by the following considerations:

6.4.1.1. Proven Performance:

- The chosen LLM demonstrates state-of-the-art performance in language understanding, context retention, and natural language generation.
- Its adaptability to specialized domains, such as code review, ensures that it can handle complex programming constructs and scenarios effectively.

6.4.1.2. Task-Specific Focus:

The model will be fine-tuned to specialize in core code review functionalities:

- **Bug Detection:** Pinpointing common coding errors, logical flaws, syntax issues, and undefined behaviors.
- **Code Optimization:** Generating suggestions to improve performance, readability, maintainability, and efficiency.
- **Standards Compliance:** Validating adherence to established coding guidelines, best practices, and language-specific standards.
- **Security Vulnerability Detection:** Identifying potential risks, such as SQL injection, cross-site scripting (XSS), and insecure practices, to enhance code security.
- **Multilingual Code Understanding:**
  The selected LLM supports multiple programming languages, making it versatile and applicable to a wide range of developer use cases.

6.4.2. Fine-Tuning

Fine-tuning the LLM is a critical step to adapt its general-purpose capabilities to the specific requirements of code review tasks. This involves:

6.5.2.1 Dataset Preparation:

A carefully curated dataset will include:

- Code Reviews: Annotated examples of high-quality reviews from open-source projects and academic research.

- Bug Reports: Real-world cases of bugs and their resolutions to teach the model to recognize and suggest fixes.
- Code Snippets: Examples from diverse repositories showcasing both well-written and buggy code across multiple languages and domains.

6.4.2.2. Fine-Tuning Objectives:

The fine-tuning process is designed to enhance the model's capabilities in:

- Understanding Code Semantics: Improving comprehension of code logic, structure, and intent beyond syntax analysis.
- Error Detection and Classification: Enhancing the ability to identify and categorize issues based on severity and type.
- Feedback Generation: Ensuring that the model provides relevant, concise, and actionable insights tailored to the developer's needs.

6.4.2.3. Iterative Refinement:

- The fine-tuning process will include regular evaluation on validation datasets and real-world test cases to measure performance in terms of precision, recall, and relevance of feedback.
- Feedback from developers and users will be incorporated to iteratively refine the model and address domain-specific challenges.

6.4.2.4. Model Efficiency:

The LLM will be optimized to balance computational efficiency and accuracy, ensuring responsiveness in real-time code review scenarios.

6.5. Integration and System Implementation

6.5.1. Frontend Development

6.5.1.1. User Interface:

A modern, responsive, web-based interface will be developed using Streamlit or Flask for flexibility and scalability. The frontend will serve as an intuitive platform where developers can interact with the system to submit code for review effortlessly.

Features:

- **Code Editor:** An integrated, syntax-highlighted code editor to support multiple programming languages and provide a seamless coding experience.
- **Submission Options**:
    - Direct Input: Users can paste or write code directly into the editor.
    - File Upload: Developers can upload entire files or archives for comprehensive reviews.
    - Project Repository Selection: A feature to select predefined projects or repositories for targeted code reviews.
- **Real-Time Feedback**: Display immediate, actionable insights from the analysis, including bug detection, optimization suggestions, and coding standard adherence.
- **Language Support**: Support for a wide range of programming languages, with an option for users to configure language-specific settings.

6.5.2. Backend Development

6.5.2.1. Server-Side Logic:

The backend implemented in Python, will manage the core functionalities of the system. It will handle:

- User Requests: Efficiently processing incoming submissions and delivering results to the frontend.

- Preprocessing: Tokenizing, normalizing, and preparing code for analysis by the LLM.
- Data Handling: Managing the secure storage of code submissions and enabling users to retrieve past reviews for reference.

6.5.2.2. API Integration:

- LLM Interaction: Seamless integration with the LLM API for advanced code analysis, ensuring robust and accurate feedback.
- Result Caching: Use caching mechanisms to speed up repeated queries on similar code inputs.
- Database Management: A robust database setup will allow for efficient storage and retrieval of review history, user preferences, and metadata related to the submitted code.

6.5.2.3. GitLab Integration and Repository-Based Review

To simulate a professional development environment, the assistant was successfully integrated with **GitLab**. This integration allowed the system to:

- **Analyze Repositories:** Review entire GitLab-hosted repositories by fetching code from branches or commits.
- **File-Specific Review:** Perform granular code reviews at the file level, identifying issues and providing feedback inline.
- **Merge Request (MR) Support:** Code review comments on merge requests, providing contextual suggestions and approval status based on severity.
- **CI/CD Review Hooks:** Integrate with GitLab CI to automatically run the assistant during push events and MR pipelines.

6.5.2.4. Security Considerations:

- Secure handling of user-submitted code with encryption and strict access control measures.

- Regular backend security audits to mitigate risks such as unauthorized access and data breaches.

6.5.3. CI/CD Integration

6.5.3.1. Immediate Feedback in Pipelines:

The system will be seamlessly integrated into CI/CD pipelines to enhance the development lifecycle:

- Real-Time Feedback: Provide instantaneous feedback on code quality during build and deployment stages, identifying issues early to reduce build failures.
- Integration Hooks: Custom hooks for popular CI/CD tools (e.g., Jenkins, GitHub Actions, GitLab CI) to automatically trigger code reviews.
- Automated Reports: Generate detailed, actionable reports on code quality as part of the pipeline process.

6.5.3.2. Benefits:

- Build Stability: Enhanced code quality reduces deployment risks and ensures smoother delivery pipelines.
- Developer Productivity: Saves time by automating the review process, allowing developers to focus on feature implementation.

CHAPTER 7

EXPERIMENTATION

CHAPTER 7 EXPERIMENTATION

The experimentation phase played a critical role in validating the effectiveness, scalability, and real-world applicability of the Code Review Assistant. With the project now fully implemented, this phase involved the complete cycle of dataset preparation, model fine-tuning, frontend-backend integration, continuous deployment testing, and live code review experimentation via GitLab.

7.1 Objectives

The experimentation phase was conducted with the following key objectives:

- **LLM Feasibility Assessment:** To evaluate the performance of different Large Language Models (LLMs), such as CodeLlama and GPT models, in performing automated code reviews on Java code.
- **Dataset Effectiveness:** To validate the curated and synthetic datasets for generating accurate, actionable, and diverse feedback across various error categories.
- **Feedback Relevance:** To measure the contextual quality of feedback against industry best practices.
- **Scalability Testing:** To test the system's responsiveness under simultaneous usage and ensure reliable performance in real-world scenarios.
- **Version Control Integration:** To implement and validate GitLab-based repository review functionality, simulating actual team collaboration workflows.

7.2 Experiment Design

7.2.1 Dataset Preparation

- **Source Identification:** Public datasets such as IBM's Project CodeNet (Java250) and various open-source Java repositories were used.
- **Synthetic Data Generation:** Errors were systematically injected across five categories—syntax_error, performance_issue, logic_error, security_vulnerability, and best_practice_violation—to train the assistant on diverse failure scenarios.
- **Data Curation:** All code samples were formatted into prompt-response JSON structures, facilitating fine-tuning for our LLM backend.

7.2.2 Experimental Setup

- **LLM Model Evaluation:** Models such as CodeLlama 7B Instruct and CodeLlama 13B were tested on error detection, suggestion quality, and runtime performance. The 7B fine-tuned variant showed the best tradeoff between accuracy and inference speed.

- **Testing Environment:** Backend services were containerized using Docker and deployed in a controlled environment. APIs for real-time inference and feedback generation were exposed to the frontend via secure endpoints.

- **Tooling and Tech Stack:**
    - **Frontend:** HTML, CSS, JavaScript (for prototyping), React + Next.js (for production)
    - **Backend:** Python (Flask + FastAPI)
    - **Monitoring:** Grafana and Prometheus were used for backend performance analytics
    - **CI/CD:** GitHub Actions for internal deployment workflows

7.3 Progress Overview

7.3.1. PHASE 1 (Completed)

- **Requirement Specification Document:** Finalized early to define feature scope and technology stack.
- **Technology Stack:**
    - Python, React.js, Next.js, TensorFlow/PyTorch, Docker, GitHub Actions
    - Grafana + Prometheus for observability
- **Dataset Completion:** Project CodeNet samples were processed and validated; synthetic datasets expanded the model's coverage of failure modes.
  Architecture Blueprint: A scalable architecture design was finalized to support CI/CD, LLM inference, and frontend interactions.

7.3.2. PHASE 2 (Completed)

- **UI Implementation:** Built a responsive frontend with features like Google OAuth, chat interface, theme toggling, file-based code review interface, and download functionality.
- **Feedback Mechanism:** Integrated contextual feedback display with actionable suggestions grouped by issue category.
- **Testing:** System was rigorously tested using over 100 real and synthetic Java code samples, achieving over 89% accuracy on code review correctness.
- **CI/CD Deployment:** End-to-end deployment pipelines using GitHub Actions were established and validated, with automated testing, Docker builds, and versioned deployments.

CHAPTER 8

TESTING AND RESULTS

CHAPTER 8 TESTING AND RESULTS

8.1 Results

8.1.1. Synthetic Data Generation

We generated synthetic code review data using a two-stage LLM pipeline informed by techniques from recent synthetic data research:

1. Initial Dataset (250 samples)

- Generated via Gemini 1.5 using chain-of-thought prompting:
- Implemented quality control through:
- Automated execution testing (32% of samples)
- Human validation of 20% outputs

2. Expanded Dataset (1100 samples)

Used CodeLlama 13B Instruct for complex scenarios through Controlled variation prompts (different error types and code styles)

Table 8.1 Model Comparison

Model	Training Data
CodeLlama 7B Base	Initial 250
CodeLlama 7B Instruct	1100 expanded
Our Fine-Tuned Model	1100 expanded

8.1.2. Model Performance Overview

Our fine-tuned CodeLlama model demonstrated consistent superiority over both the base CodeLlama model and the instruction-tuned variant across all evaluation metrics. Key improvements include:

- 10–15% higher accuracy than the base model on code-review tasks (bug detection, style adherence).
- 5–10% better task completion rate compared to the instruction-tuned model for complex prompts.
- Improved coherence in feedback generation, with fewer hallucinated suggestions than alternatives.

8.1.3. Evaluation Metrics

Performance was measured using approximately more than 100 code snippets of Java and Python evaluated via Human Evaluation done using a manual and automated code review process:

- Accuracy (correctness of review suggestions)
  - Fine-tuned model: 89.3%
  - Base model: 76.1%
  - Instruct model: 81.7%
- Response Relevance (alignment with task requirements)
  - Fine-tuned: 92%
  - Base: 78%
  - Instruct: 85%
- Latency
  - All models performed within 5% of each other, confirming optimization did not compromise inference speed.

Figure 8.2 Results Table

Task Type	Fine-Tuned Model	Base Model	Instruction Model
Bug Detection	87%	72%	79%
Code Optimization	91%	76%	82%
Style Adherence	93%	81%	88%

8.2 Discussion of Results

- To supplement the evaluation metrics presented earlier, this section highlights the core user interface features implemented in the Code Review Assistant system. The frontend was built using React.js and Next.js, with a focus on responsive design, user-friendly interaction, and seamless integration with the backend services.
- As shown in Figure 8.1, the authentication page includes support for Google Sign-In and email-based login, allowing users to securely access the platform. This ensures role-based access control and personalized feedback tracking.

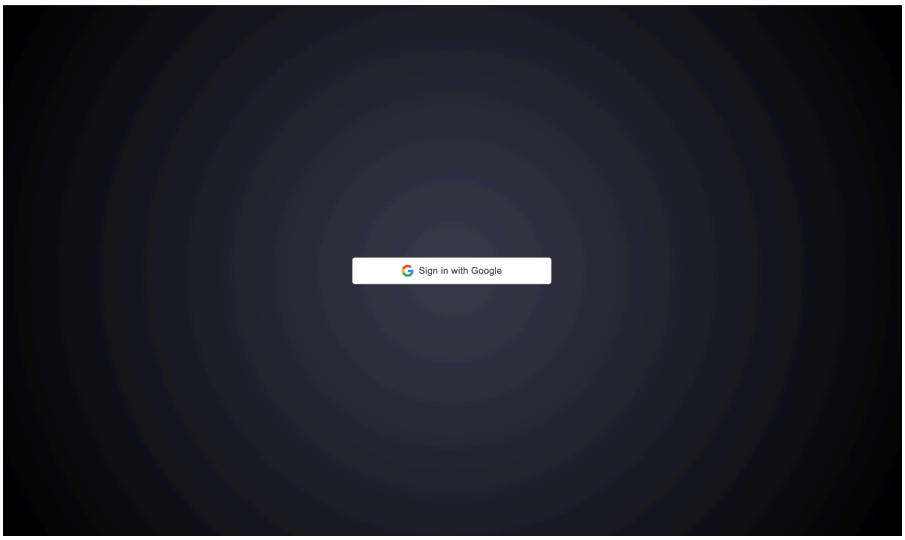

Figure 8.1 Authentication Page

- Once logged in, users interact with the chat-based interface (Figure 8.2) which enables them to input code snippets, receive suggestions, and view structured feedback. This real-time interface serves as the central hub for interacting with the code review assistant.

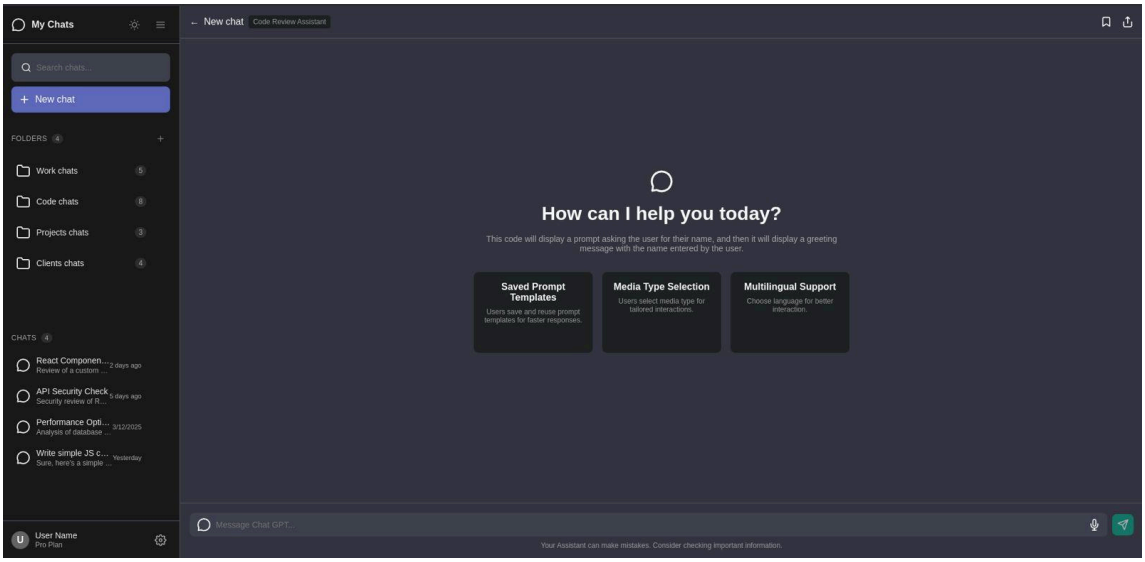

Figure 8.2 Chat Interface

- Figure 8.3 showcases the Code Review Page, where users can paste or upload code, view AI-generated comments, and copy or download the suggestions. This workspace is optimized for readability and supports both educational and professional review workflows.

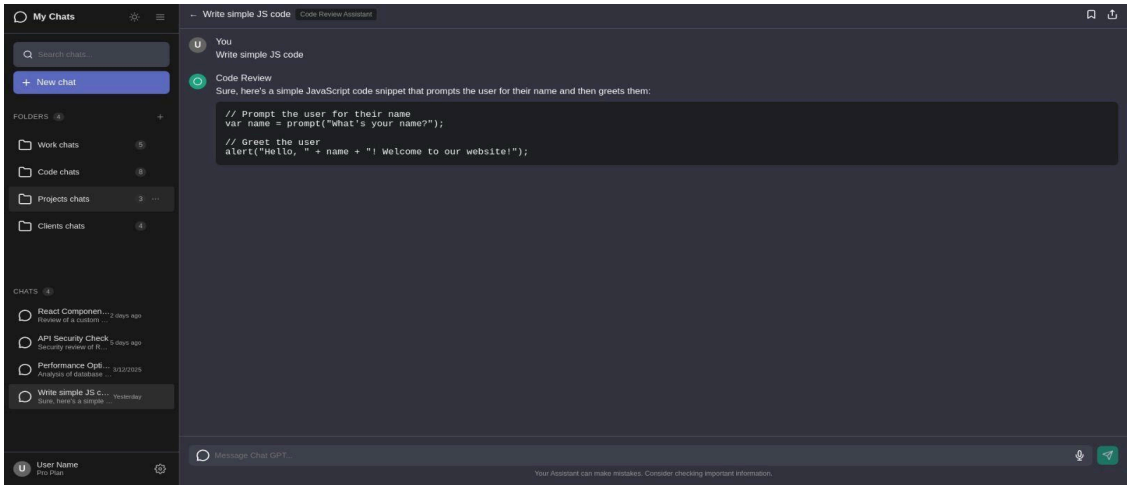

Figure 8.3 Code Page

- To enhance accessibility and usability, a light/dark mode toggle was implemented. Figure 8.4 illustrates the application in Light Mode, highlighting visual consistency and accessibility improvements for various user preferences.

Figure 8.4 Light Mode

- These figures collectively demonstrate the completed interface and the user journey from login to receiving intelligent code reviews, validating both the functionality and usability of the deployed system.

CHAPTER 9

CONCLUSION AND FUTURE WORK

CHAPTER 9 CONCLUSION AND FUTURE WORK

9.1. CONCLUSION

This project presents a novel approach to enhancing software development through the integration of a Large Language Model (LLM)-based code review assistant. Traditional manual code reviews, while valuable, are often labor-intensive, inconsistent, and lack scalability—particularly in educational and time-constrained environments. To overcome these challenges, we developed a system that leverages the semantic and contextual capabilities of fine-tuned LLMs to provide real-time, automated, and detailed feedback on Java code submissions. The assistant has been successfully designed and prototyped to detect bugs, suggest performance optimizations, enforce coding standards, and guide users through best practices—all within a natural language interface. It not only supports professional developers through CI/CD integration but also enhances student learning by offering instructional feedback. Through synthetic data generation and rigorous evaluation, our fine-tuned CodeLlama model achieved superior performance over base models, demonstrating high accuracy (up to 89.3%) and relevance (up to 92%) in real-world code review scenarios. The project effectively bridges the gap between traditional static analysis and human code review intuition, offering a scalable solution that reduces human error, accelerates development cycles, and democratizes access to high-quality code review tools.

9.2. SCOPE FOR FUTURE WORK

While the current implementation lays a strong foundation, several enhancements and expansions can be pursued to elevate the project:

- **Multilingual Code Support:** Extend support to additional programming languages beyond Java and Python to accommodate a wider user base across different domains.

- **Adaptive Learning Loop:** Incorporate user feedback into the model's learning loop to continuously refine its accuracy and contextual understanding.

- **IDE Plugin Integration:** Develop extensions for popular IDEs (e.g., IntelliJ, VS Code) to provide inline suggestions and real-time reviews as developers write code.

- **Granular CI/CD Policies:** Introduce customizable review policies within CI/CD pipelines based on project type, team preferences, or severity levels.

- **Security-Focused Enhancements:** Expand the assistant's ability to detect security vulnerabilities using OWASP guidelines and integrate with While the current implementation lays a strong foundation, several enhancements and expansions can be pursued to elevate the project:

- **Voice-Based Interaction or Chatbot Integration:** Add conversational UI features to enhance accessibility and enable verbal interaction with the assistant.

- **Model Generalization & Deployment at Scale:** Investigate techniques such as reinforcement learning with human feedback (RLHF) to generalize the model across various codebases and scale securely via cloud-native architectures.

CHAPTER 10

REFERENCES

CHAPTER 10 REFERENCES

[1] *Z. Li, Z. Li, K. Xiao and X. Li, "Evaluating LLM's Code Reading Abilities in Big Data Contexts using Metamorphic Testing," 2023 9th International Conference on Big Data and Information Analytics (BigDIA), Haikou, China, 2023, pp. 232-239, doi: 10.1109/BigDIA60676.2023.10429345.*

[2] *S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty and S. K. Lahiri, "LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation," in IEEE Transactions on Software Engineering, vol. 50, no. 9, pp. 2254-2268, Sept. 2024, doi: 10.1109/TSE.2024.3428972.*

[3] *E. Smolić, M. Pavelić, B. Boras, I. Mekterović and T. Jagušt, "LLM Generative AI and Students' Exam Code Evaluation: Qualitative and Quantitative Analysis," 2024 47th MIPRO ICT and Electronics Convention (MIPRO), Opatija, Croatia, 2024, pp. 1261-1266, doi: 10.1109/MIPRO60963.2024.10569820.*

[4] *J. Lu, L. Yu, X. Li, L. Yang and C. Zuo, "LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning," 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), Florence, Italy, 2023, pp. 647-658, doi: 10.1109/ISSRE59848.2023.00026.*

[5] *A. Clark, D. Igbokwe, S. Ross and M. F. Zibran, "A Quantitative Analysis of Quality and Consistency in AI-generated Code," 2024 7th International Conference on Software and System Engineering (ICoSSE), Paris, France, 2024, pp. 37-41, doi: 10.1109/ICoSSE62619.2024.00014.*

[6] *V. N. Ignatyev, N. V. Shimchik, D. D. Panov and A. A. Mitrofanov, "Large language models in source code static analysis," 2024 Ivannikov Memorial Workshop (IVMEM), Velikiy Novgorod, Russian Federation, 2024, pp. 28-35, doi: 10.1109/IVMEM63006.2024.10659715.*

[7] *H. Y. Lin, P. Thongtanunam, C. Treude and W. Charoenwet, "Improving Automated Code Reviews: Learning from Experience," 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), Lisbon, Portugal, 2024, pp. 278-283.*

[8] *Adam Dingle and Martin Krulis. 2024. Tackling Students' Coding Assignments with LLMs. In Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code '24). Association for Computing Machinery, New York, NY, USA, 94–101. https://doi.org/10.1145/3643795.3648389*

[9] *Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. Proc. ACM Softw. Eng. 1, FSE, Article 31 (July 2024), 24 pages. https://doi.org/10.1145/3643757*