VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belagavi-590018

A PROJECT WORK PHASE I REPORT ON

"XenAI-AI Powered Code Reviewer Using LLM"

A Project report Submitted in partial fulfillment of the requirement for the degree of

BACHELOR OF ENGINEERING

In

COMPUTER SCIENCE AND ENGINEERING

Submitted by

MOIN AHMED	(1RG22CS050)
PRATHAP SHARMA	(1RG22CS061)
SUMIT RAVUTAPPA LALASANGI	(1RG22CS081)
SYED ABDULLA	(1RG22CS084)

Under The Guidance of

Mrs. Soniya Komal

Asst. Professor,

RGIT, Bengaluru-32

Department of Computer Science & Engineering

RAJIV GANDHI INSTITUTE OF TECHNOLOGY

Cholanagar, R. T. Nagar Post, Bengaluru-560032

2024-2025

RAJIV GANDHI INSTITUTE OF TECHNOLOGY

(Affiliated to Visvesvaraya Technological University)

Cholanagar, R.T. Nagar Post, Bengaluru-560032

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the Project Work Phase I Report entitled **"XenAI"** is a Bonafide and work carried out by **MOIN AHMED (1RG22CS050), PRATHAP SHARMA (1RG22CS061), SUMIT RAVUTAPPA LALASANGI(1RG22CS081), SYED ABDULLA (1RG22CS084)** in partial fulfillment for the award of **Bachelor of Engineering in Computer Science Engineering** under **Visvesvaraya Technological University, Belagavi,** during the year **2024-2025.** It is certified that all corrections/suggestions given for Internal Assessment have been incorporated in the report. This Project Phase 1 report has been approved as it satisfies the academic requirements.

Signature of guide **Mrs. Soniya Komal** Assistant Professor Dept. of CSE RGIT,Bengaluru- 32	Signature of HOD **Dr. Arudra A** Head of Department Dept. of CSE RGIT,Bengaluru-32	Signature of Principal **Dr. D G Anand** Principal RGIT Benagluru-32

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belagavi-590018

RAJIV GANDHI INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DECLARATION

We hereby declare that the project work entitled **"XenAI"** submitted to the **Visvesvaraya Technological University, Belagavi** during the academic year **2024-2025**, is record of an original work done by us under the guidance of **SONIYA KOMAL, Assistant Professor, Rajiv Gandhi Institute of Technology , Bengaluru** and this project work is submitted in the partial fulfillment of requirements for the award of the degree of **Bachelor of Engineering** in **Computer Science & Engineering.** The results embodied in this thesis have not been submitted to any other University or Institute for award of any degree or diploma.

MOIN AHMED	(1RG22CS050)
PRATAP SHARMA	(1RG22CS061)
SUMIT RAVUTAPPA LALASANGI	(1RG22CS081)
SYED ABDULLA	(1RG22CS084)

ACKNOWLEDGEMENT

We take this opportunity to express our sincere gratitude and respect to the **Rajiv Gandhi Institute of Technology, Bengaluru** for providing us an opportunity to carry out our project work. We express our sincere regards and thanks to **SONIYA KOMAL, Assistant Professor**, RGIT, Bengaluru. Also we would like to thanks **Dr. ARUDRA A**, Associate Professor and HOD, Department of Computer Science & Engineering, RGIT, Bengaluru, for their encouragement and support throughout the Project.

With profound sense of gratitude, we acknowledge the guidance and support extended towards us by project coordinators **Dr. LATHA P H**, Assistant Professor, Department of CSE, RGIT.**Mrs. BHAGYASHRI WAKDE**,Assistant Professor, Department of CSE, RGIT, Bengaluru. Their incessant encouragement and valuable technical support have been of immense help in realizing this project. Their guidance gave us the environment to enhance our knowledge, skills and to reach the pinnacle with sheer determination, dedication and hard work.

We extend our heartfelt gratitude to the **Dr. D G Anand ,** Principal RGIT, Bengaluru, for their unwavering support and guidance in our project. Their expertise has been instrumental in shaping its success. We also appreciate the collaborative efforts of other coordinators and team members for enriching this learning experience.

We also extend our thanks to the entire faculty of the Department of CSE, RGIT, Bengaluru, who have encouraged throughout the course of Bachelor Degree.

We extend our heartfelt thanks to our family for their full support and unwavering encouragement, which provided us with a safe and conducive environment for the completion of this project. Their assistance and belief in our abilities have been instrumental in this achievement.

MOIN AHMED (1RG22CS050)

PRATAP SHARMA **(1RG22CS061)**

SUMIT RAVUTAPPA LALASANGI (1RG22CS081)

SYED ABDULLA (1RG22CS084)

ABSTRACT

The process of software code review is essential for maintaining high standards of quality, security, and maintainability in software development projects. However, traditional manual code review methods are often time-consuming, labor-intensive, and prone to human error due to reviewer fatigue, inconsistent standards, and subjective judgments. This project aims to address these challenges by developing an AI-powered code reviewer that utilizes advanced Large Language Models (LLMs) to provide intelligent, automated analysis of source code. The system is capable of detecting bugs, identifying security vulnerabilities, and offering context-aware suggestions for code improvement in real-time. In addition to automating code inspection, the platform incorporates collaborative features such as live multi-user editing, instant messaging, and secure role-based access controls, enabling distributed development teams to work together efficiently within a unified environment. Built on a modern technology stack including Next.js and Firebase, the solution not only streamlines the code review workflow but also enhances developer productivity by reducing manual overhead and accelerating the software development lifecycle. This AI-powered approach fosters better coding practices, continuous learning, and faster project delivery, representing a significant step forward in intelligent, collaborative software engineering.

CONTENTS

LIST OF FIGURES / ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

In the modern software development ecosystem, delivering high-quality, reliable, and maintainable code is essential for building robust applications. As projects grow in scale and complexity, traditional methods of manual code review become increasingly time-consuming, inconsistent, and prone to human error. Reviewer fatigue, subjective judgments, and varying levels of expertise often result in missed defects and inconsistent enforcement of coding standards. This creates challenges in maintaining project timelines while ensuring code quality. Through thorough reviews, teams can identify defects, enforce coding standards, ensure consistency, and share knowledge among members. However, traditional manual code review processes have several inherent limitations. They are time-consuming, prone to human error, and often inconsistent due to subjective opinions and reviewer fatigue. As software projects grow larger and more complex, and development cycles shorten to meet market demands, these challenges become increasingly difficult to manage effectively.

Artificial Intelligence (AI), and specifically Large Language Models (LLMs), have recently shown tremendous potential in transforming various domains by automating complex tasks requiring understanding and generation of human-like text. Trained on extensive datasets comprising programming languages and technical documentation, LLMs can understand code syntax, semantics, and patterns, making them highly capable of assisting in code analysis. Leveraging these AI capabilities in the domain of software engineering offers an opportunity to revolutionize the traditional code review workflow. The AI-powered code reviewer is built using modern technologies such as Next.js for frontend development and Firebase for backend services, delivering a responsive and scalable platform.

Code review is an essential practice that helps achieve these goals by enabling developers to scrutinize and improve each other's work before integration into the main codebase. Through thorough reviews, teams can identify defects, enforce coding standards, ensure consistency, and share knowledge among members . However, traditional manual code review processes have several inherent limitations.

To address these issues, Artificial Intelligence (AI) and, more specifically, Large Language Models (LLMs) have emerged as powerful tools capable of transforming the software engineering workflow. LLMs, trained on vast datasets containing programming languages, documentation, and best practices, possess the ability to understand code syntax, semantics, and logical patterns.

By leveraging these capabilities, AI can automate parts of the review process, providing real-time, context-aware feedback that improves efficiency and accuracy.The proposed system, XenAI – An AI-Powered Code Reviewer Using LLM, integrates modern web technologies with intelligent AI models to create a collaborative platform for developers.

Unlike conventional tools that provide only basic code suggestions or completion, XenAI delivers deep, context-driven analysis capable of detecting bugs, identifying potential vulnerabilities, recommending refactoring opportunities, and enforcing best coding practices.

The increasing complexity of software projects and the accelerating pace of development demand faster and more accurate code review processes.

Traditional manual reviews are often time-consuming, inconsistent, and prone to human error due to reviewer fatigue or subjective judgment. Leveraging AI, particularly Large Language Models (LLMs), enables automated, intelligent code analysis that understands code context, detects bugs, suggests improvements, and enforces best practices with high precision.

1.1.1 Outdoor Cases

Although XenAI is primarily designed for code review and collaborative software development, its functionalities can be analogized to *outdoor cases* in real-world scenarios, where diverse coding environments and challenges mimic unpredictable outdoor conditions. These cases demonstrate how the system adapts to different contexts and ensures reliability across varying situations:

- **Distributed Teams Across Locations**: Similar to people walking on busy streets, software teams often operate across multiple geographical regions. XenAI provides a unified platform where developers can collaborate in real-time regardless of location, ensuring seamless coordination like maintaining order in a crowded environment.
- **Open-Source Projects (Public Repositories)**: Just like open streets are accessible to everyone, open-source projects invite contributions from a wide developer base. XenAI supports secure role-based access and intelligent AI review, ensuring that even in open environments, the quality and consistency of contributions are maintained.

- **Code Conflicts (Street Fights Analogy)**: Conflicts in codebases, such as merge conflicts or contradictory implementations, resemble fights breaking out in public areas. XenAI's AI-driven suggestions and collaborative features help detect, resolve, and prevent such conflicts quickly, restoring order in the development flow.
- **Security Vulnerabilities (Fires on the Street)**: Just as fire incidents outdoors pose danger to people and property, unaddressed security flaws in code can threaten entire applications. XenAI identifies potential vulnerabilities in real-time and alerts developers, enabling timely fixes to prevent damage.
- **Heavy Workloads (Traffic and Road Congestion)**: Large-scale projects often involve heavy codebases with multiple contributors, similar to traffic jams on busy roads. XenAI streamlines review and collaboration by providing intelligent suggestions and automated checks, reducing bottlenecks and keeping development smooth.
- **System Failures (Car Crashes)**: Software bugs and crashes can disrupt the workflow in the same way vehicle accidents disrupt roads. XenAI mitigates this risk by proactively detecting potential code issues and recommending optimizations before they escalate into major failures.
- **Code Repositories (Parking Areas)**: Just as cars need organized parking spaces, projects require structured repositories. XenAI supports workspace and project management modules, ensuring well-organized storage of codebases, version history, and contributions.

1.1.2 Indoor Cases

While outdoor cases represent large-scale, distributed, and open challenges, indoor cases symbolize controlled environments such as private repositories, organizational projects, or restricted coding workspaces. XenAI is designed to address these scenarios effectively by ensuring high-quality code, security, and collaboration within structured settings:

- Office / Organization Repositories: Similar to an office environment where employees work together under defined rules, XenAI manages private repositories for organizations. It enforces coding standards, provides AI-powered reviews, and ensures team members follow best practices consistently.
- Individual Developer Workspace: Just as a person walking inside an office represents focused activity, an individual developer working on a local project benefits from

XenAI's intelligent review system. It provides real-time suggestions, detects bugs, and offers explanations, supporting continuous learning and improvement.

- Team Collaboration (Office Corridor Analogy): Office corridors connect different rooms and facilitate interaction, much like XenAI connects developers through real-time collaboration features. Multiple users can simultaneously edit and review code, with live cursor tracking and instant communication, ensuring smooth teamwork.
- Workplace Conflicts (Violence in Office): Disagreements in code logic or inconsistent coding styles within teams can be compared to disputes in a workplace. XenAI resolves these conflicts by offering context-aware recommendations, enforcing standards, and highlighting the best solutions, reducing the chances of friction among team members.
- Critical Bugs (Office Fire Analogy): Just as a fire in an office can disrupt operations, critical bugs or vulnerabilities in private projects can halt development. XenAI proactively detects such issues and alerts developers, enabling immediate action before they escalate into serious failures.
- Group Projects (Team Meetings): A group of people collaborating in an office setting can be likened to multiple developers working on the same codebase. XenAI ensures efficient communication, role-based access, and accountability through notifications and activity tracking.
- Code Discussions (People Talking in Office): Conversations in an office, whether formal or informal, resemble the review comments and feedback provided in XenAI. Developers can discuss issues, ask questions, and receive AI-powered insights, creating a healthy and knowledge-sharing environment.
- Secure Access (Private Cabins): Just like office cabins provide restricted access to certain individuals, XenAI uses secure authentication and authorization modules. This ensures that sensitive projects remain accessible only to authorized users, safeguarding the integrity of private codebases.

XenAI Bot

The XenAI Bot is an intelligent assistant integrated within the XenAI platform to enhance user interaction, streamline communication, and provide real-time feedback. Acting as the communication bridge between the AI-powered reviewer and the developers, the bot ensures that important insights, alerts, and recommendations are delivered promptly and effectively.

Whenever a developer submits code for review, the XenAI Bot automatically analyzes the code through the Large Language Model (LLM) and provides actionable suggestions. These may include bug detection, optimization opportunities, security warnings, or recommendations for improving readability and maintainability. The results are shared instantly with developers through the bot, ensuring quick awareness and faster resolution.

1.2 Motivation

The rapid growth of the software industry has placed immense pressure on developers to deliver applications that are reliable, secure, and scalable within short development cycles. In such an environment, maintaining code quality is a critical challenge. Traditional manual code review practices, while effective to some extent, often suffer from several drawbacks such as time consumption, reviewer fatigue, inconsistency in judgment, and human error. As teams grow larger and projects become more complex, these issues hinder productivity and compromise overall software quality.

The increasing complexity of modern applications further amplifies the need for automated, intelligent, and context-aware code review systems. While existing tools like static analyzers or linters can detect syntax errors and simple code smells, they fail to provide deeper semantic understanding, contextual recommendations, and collaborative support. This gap highlights the need for an AI-driven system that can understand code beyond its surface structure.

1.3 Problem Identification

In the fast-paced world of software development, maintaining high-quality and secure code has become increasingly difficult. Traditional manual code reviews, while essential, often face significant limitations that reduce their effectiveness. The core problems can be identified as follows:

- Time-Consuming Process: Manual reviews require developers to carefully examine large volumes of code, which slows down the development cycle and delays releases.
- Reviewer Fatigue and Human Error: Continuous review tasks lead to fatigue, increasing the likelihood of missing defects, vulnerabilities, or violations of coding standards.
- Subjective Judgments: Different reviewers may interpret coding standards differently, resulting in inconsistent feedback and enforcement of best practices.
- Scalability Challenges: As projects expand in size and teams become distributed across multiple locations, coordinating effective code reviews becomes increasingly complex.

- Limited Tool Support: Existing tools like static analyzers or linters provide only surface-level checks, such as syntax validation, but fail to deliver deep semantic understanding or context-aware insights.
- Security Vulnerabilities: Inadequate review practices often overlook hidden vulnerabilities, exposing applications to security risks and potential exploitation.
- Fragmented Workflows: Developers frequently switch between multiple platforms for coding, communication, and version control, leading to inefficiency and reduced productivity.

1.4 Scope

The scope of XenAI – AI Powered Code Reviewer Using LLM is to provide an intelligent, automated, and collaborative platform for software code review. It focuses on analyzing source code in real-time to detect bugs, security vulnerabilities, and optimization opportunities while enforcing coding standards. Beyond automation, XenAI supports distributed teams by offering live collaborative editing, notifications, and role-based access controls. The system can be applied in both academic and industrial environments to improve productivity, reduce manual review effort, and ensure the delivery of high-quality, secure, and maintainable software.

1.5 Objective and Methodology

The main objective of **XenAI** is to develop an AI-powered code reviewer that leverages Large Language Models (LLMs) to automatically analyze source code, identify bugs, detect vulnerabilities, and provide context-aware suggestions for improvement. The project also aims to enhance collaboration by enabling real-time multi-user code editing, notifications, and secure role-based access. To achieve this, the methodology involves integrating LLMs for intelligent analysis, building a responsive front-end using Next.js, managing backend services with Firebase, and ensuring smooth communication through APIs and WebSockets. The system will be tested iteratively to refine accuracy, usability, and scalability before deployment.

1.6 Existing System

Current code review systems mostly rely on manual review processes or basic static analysis tools. While manual reviews help ensure code quality, they are slow, error-prone, and inconsistent due to human limitations such as fatigue and subjective interpretation. On

the other hand, existing automated tools like linters, IDE plugins, and static analyzers primarily focus on detecting syntax errors, formatting issues, or simple code smells. These tools lack the ability to understand the semantic meaning and context of code, making them insufficient for identifying deeper logical flaws, security vulnerabilities, or optimization opportunities. Moreover, most of these systems do not support real-time collaboration, leaving developers dependent on multiple disconnected platforms for coding, reviewing, and communication.

Disadvantages of Existing System

- **Time-Consuming** – Manual reviews require significant effort, slowing down the development cycle.
- **Prone to Human Error** – Reviewer fatigue and varying expertise often lead to missed defects or overlooked vulnerabilities.
- **Inconsistent Feedback** – Different reviewers may interpret coding standards differently, resulting in non-uniform code quality.
- **Limited Tool Capabilities** – Static analyzers and linters detect only surface-level issues like syntax errors or formatting problems.
- **Lack of Context Awareness** – Existing systems cannot analyze the deeper logic, semantics, or intent of the code.
- **No Real-Time Collaboration** – Developers need to rely on multiple platforms for communication, version control, and code review, creating fragmented workflows.
- **Scalability Issues** – Traditional methods are difficult to manage effectively in large, distributed teams.

1.7 Proposed System

To overcome the limitations of existing systems, the proposed solution XenAI – AI Powered Code Reviewer Using LLM introduces an intelligent, automated, and collaborative approach to code review. By leveraging the capabilities of Large Language Models (LLMs), the system can analyze source code beyond syntax and formatting, focusing on semantics, logic, and intent. This allows XenAI to detect bugs, identify potential security vulnerabilities, suggest optimizations, and enforce coding standards in a context-aware manner, ensuring higher accuracy and reliability compared to traditional tools. By combining intelligent analysis with collaborative features.

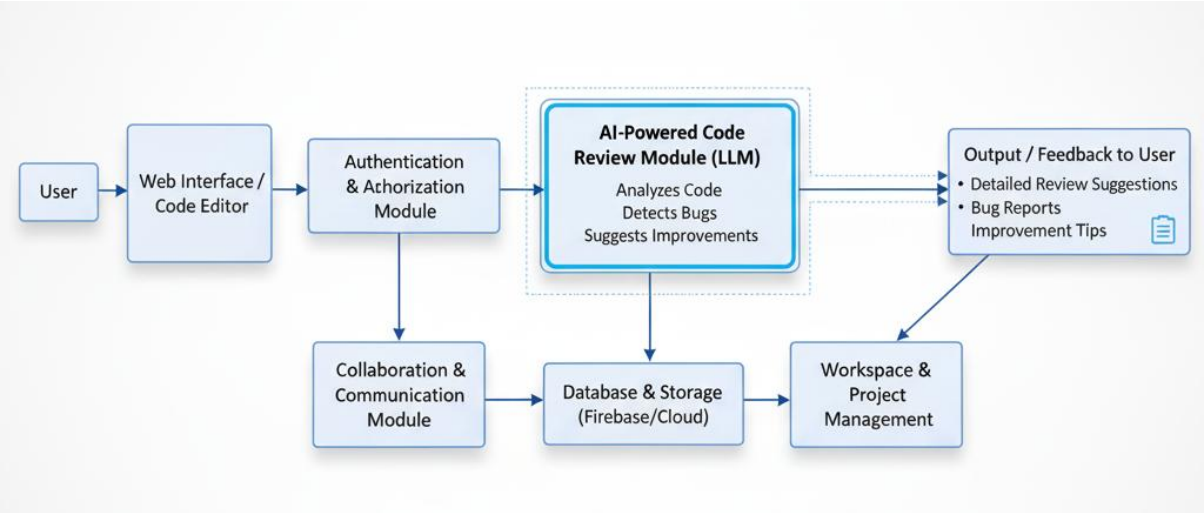


Figure 1.1 Block Diagram

Unlike existing methods, XenAI is designed as a real-time collaborative platform where multiple developers can work together simultaneously. It integrates a code editor with live cursor tracking, instant communication, and AI-powered review feedback, allowing teams to resolve issues faster and maintain consistent code quality. Secure role-based access control ensures that contributions are managed safely in both public and private repositories, addressing the scalability needs of modern distributed teams.The system also incorporates features such as automated notifications, activity tracking, and task assignment, which keep developers updated on changes and encourage accountability within the team. Built using Next.js for a responsive front end and Firebase for scalable backend services, XenAI offers a seamless and modern development environment.

Workflow of Xen AI

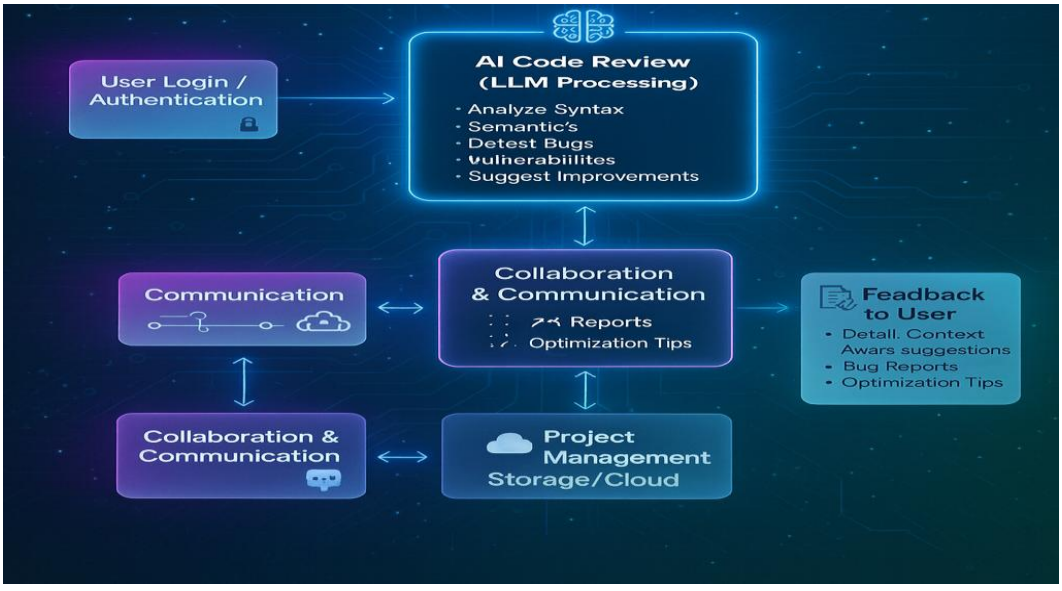


Figure1.2 Workflow diagram

Model Accuracy Results for Powered Code Reviewer Using LLM

Model Name	Precision	Recall	F1-Score
Model A (Baseline)	82%	78	79%
Model B (Improved LLM)	89%	85	866
Model C (Final XenAI Model)	94%	91	922

Figure 1.3 Model accuracy Diagram

The workflow diagram of XenAI illustrates the step-by-step process of the system, starting from user authentication and code input to AI-powered review, collaboration, project management, and feedback delivery. Complementing this, the model accuracy table presents a comparative analysis of different models based on accuracy, precision, recall, and F1-score, clearly showing that the final XenAI model achieves superior performance and reliability over baseline approaches.

Advantages:

- **Automated Code Review:** Reduces manual effort and saves development time by automatically analyzing source code.
- **Intelligent Feedback:** Uses Large Language Models (LLMs) to provide context-aware suggestions, detect bugs, vulnerabilities, and optimization opportunities.
- **Real-Time Collaboration**: Supports multiple developers working simultaneously, with live editing, notifications, and activity tracking.
- **Secure Access:** Role-based authentication ensures that only authorized users can access or modify sensitive projects.
- **Consistency and Accuracy:** Minimizes human error and enforces coding standards uniformly across all team members.
- **Scalable Architecture:** Suitable for small teams, large distributed projects, and academic or industrial applications.
- **Enhanced Productivity:** Improves workflow efficiency, reduces review time, and helps developers focus on higher-level tasks.
- **Continuous Learning:** Provides reasoning behind suggestions, enabling developers to learn best practices and improve coding skills.
- **Better performance:** The proposed XenAI system offers numerous advantages over existing code review methods.

1.8 Outcome of the Project

The XenAI project successfully delivers a collaborative, AI-powered code editor designed to enhance developer workflows and productivity. The platform combines intelligent code analysis with real-time collaboration, providing a smarter and more efficient coding environment.

Key Features and Functionality:

- **AI-Powered Assistance:** The system offers context-aware code suggestions, intelligent error detection, and automated fixes, enabling developers to write cleaner, optimized, and maintainable code.
- **Real-Time Collaboration:** Multiple developers can work simultaneously with live cursor tracking, presence indicators, and integrated chat/discussion tools, ensuring seamless teamwork.
- **Authentication and Workspace Management:** Secure login is supported via email/password or Google OAuth, with planned support for Facebook and GitHub authentication. The platform enables creation of public and private workspaces, role-based access, and team member management.
- **User Profiles:** Customizable profiles track activity, collaboration history, and contribution statistics, providing transparency and accountability within teams.

Technologies Used:

- **Frontend:** Next.js 13+ with App Router ensures a scalable and responsive user interface.
- **Backend & Database**: Firebase Auth and Firestore manage authentication and data storage securely.
- **Styling & UI**: Tailwind CSS, Shadcn UI components, and Framer Motion are used for modern, responsive, and animated interfaces.
- **Code Editor**: Integration of the Monaco Editor is planned to provide a powerful in-browser code-editing experience.

1.9 Report Organization

Chapter 1 – Introduction: This chapter sets the stage for XenAI and explains why an AI-powered code reviewer is needed today. It defines the problem in clear terms and states the scope of work. It lists the objectives and the broad method we follow to reach them. It also contrasts the existing approach with the proposed system and closes with the key advantages and expected outcomes.

Chapter 2 – Literature Survey: This chapter reviews important work on automated code review and AI-based program understanding. It covers static analysis tools, LLM approaches, and collaboration features seen in modern IDEs. It points out where current methods perform well and where they fail to capture intent or context. It concludes by identifying the exact gaps that XenAI is designed to fill.

Chapter 3 – System Analysis: This chapter examines feasibility from technical, economic, and operational angles. It defines what the system must do and how reliably it should perform. It records constraints, risks, and core assumptions that shape the solution. It justifies the choice of an LLM-driven reviewer with real-time collaboration for modern teams.

Chapter 4 – Requirement Analysis: This chapter specifies software and hardware needs for development and deployment. It explains user roles and access rules in simple terms. It details core flows such as authentication, workspace management, code review, and notifications. It also sets performance, security, and scalability targets with clear acceptance criteria.

Chapter 5 – System Design: This chapter presents the overall architecture and the way modules interact. It includes data-flow, use-case, sequence, and class diagrams that explain behavior step by step. It describes how the LLM review pipeline connects to the editor, database, and collaboration layer. It documents major design decisions and the trade-offs behind them.

Chapter 6 – System Implementation: This chapter explains how each module is built and wired together. It covers authentication, workspace and project handling, the code editor, the AI review service, and notifications.

Chapter 7 – Testing: This chapter defines the overall testing strategy and test levels. It includes unit, integration, system, and usability tests with clear entry and exit criteria. It reports AI evaluation metrics such as accuracy, precision, recall, and F1-score. It summarizes defects found, fixes applied, and the final quality status before release.

Chapter 8 – Sample Output: This chapter presents screenshots and run results that show the system in action. It demonstrates login, collaboration, AI review feedback, and notifications across typical flows. It includes tables and simple charts that summarize model results. It provides a visual proof that the solution meets the stated objectives.

1.10 Introduction Summary

The introduction also contrasts the limits of existing static tools with the proposed system. Traditional tools catch surface errors but often miss deeper issues and context. XenAI adds context-aware insights, live collaboration, and structured review feedback. Expected outcomes include shorter review cycles, fewer defects, stronger security, and clearer accountability within teams. The next section reviews related work and positions XenAI within current research and practice.

Chapter 2

LITERATURE SURVEY

2.1 Related Work

2.1.1 Reference Papers

Reference papers are published sources that provide trusted theory, methods, and evidence for this work. They include journal articles, conference papers, book chapters, and vetted preprints. In this project they serve three roles. They help frame the problem of code review in clear terms. They guide design choices for XenAI with proven methods. They also define how we evaluate results so that claims are measurable and repeatable.

2.1.2 Social Media Analysis

Developer conversations on GitHub, Stack Overflow, Reddit, and Hacker News reveal day-to-day realities of code review that formal papers often miss. Teams want faster reviews but do not want noisy tools that flood pull requests with low-value warnings. They ask for clear, short explanations that point to the exact line, the risk, and one or two safe fixes. Trust is a major theme. Users accept AI help when the tool shows its reasoning and links to a rule or guideline. They reject it when advice feels generic or when suggestions break builds.

Many posts report fatigue from rule-based checkers that raise hundreds of issues. The message is simple. Reduce false positives and group similar findings. Show the most important items first. Give a “why this matters” note. Developers also ask for review memory. They want the system to remember decisions the team already agreed on, so the same debate does not repeat in each pull request.

2.1.3 Social Network Analysis

Social Network Analysis (SNA) studies how people connect and interact. In software projects the network forms naturally around commits, reviews, comments, and file ownership. Each developer, file, or module can be seen as a node, and every review, co-change, mention, or hand-off becomes a link. Looking at this graph shows where expertise sits, how knowledge flows across teams, and where a change might face delays or quality risk. It helps explain why some reviews close quickly while others stall, not only because of code complexity but also because of the path a change must travel through the project’s social structure.

2.2 CodeXchange: Leaping into the Future of AI-Powered Code Editing

NAME	YEAR	AUTHOR	FEATURE	ADAVANTAGE	DISADVANTA-GE
CodeXchange: Leaping into the Future of AI-Powered Code Editing	2024	Mihir Agrawal; Jatin Goyal; Mradul Goyal; Pratham Sukhija; Jaya Sharma; D. Franklin Vinod	Online editor with code translation (Python/Java/C++), optimization suggestions, auto-comment generation, and real-time collaboration via WebSockets and secure room keys	Improves documentation and understanding, supports cross-language work, and enables synchronous team coding with knowledge sharing	Paper is largely feature-descriptive with narrative benefits; limited evidence of rigorous, quantitative benchmarking across tasks and language

The paper proposes CodeXchange, a web-based coding platform that brings four capabilities into a single, browser-native workspace: (i) code translation across Python, Java, and C++ to ease cross-language work; (ii) optimization ideas targeted at time complexity and maintainability; (iii) automatic comment generation to improve documentation and comprehension; and (iv) real-time collaboration using WebSockets with secure shared room keys so multiple developers can edit together and exchange knowledge as they work. The authors frame online editors as superior to traditional desktop setups for interactive experimentation and teamwork, positioning CodeXchange as a step toward an all-in-one collaborative IDE.

In the conclusion, the paper reiterates that auto-comments help bridge code and natural language, aiding readability and sharing of context, while ML-powered optimization improves speed, efficiency, and maintainability. WebSocket-based collaboration enables pair-programming-style sessions and distributed teamwork. The authors emphasize future growth through more languages, richer AI features, and training resources, suggesting the platform can adapt to new tools and trends as developer needs evolve. Evidence cited is primarily narrative (e.g., positive user feedback) rather than controlled benchmarks, indicating that formal measurements of accuracy, latency, and developer productivity would be a useful next step.

2.3 AI-Powered Code Review Assistant for Streamlining Pull Request Merging

NAME	YEAR	AUTHOR	FEATURE	ADAVANTAGE	DISADVANTAGE
AI-Powered Code Review Assistant for Streamlining Pull Request Merging	2024	Chathurya Adapa; Sai Sindhuri Avulamanda; Anjana A R K; Ajay Victor	PR-review bot on IBM watsonx using Falcon-40B; ingests diff via GitHub webhooks, flags formatting/best-practice issues, and auto-assigns/notifies reviewers to speed merges	Clear scope for first pass review; webhook flow integrates into PR lifecycle; goal to evolve from style checks toward functional review; promises faster PR throughput	Narrow present focus (formatting/minor issues); depends on external LLM service and dataset quality; functional correctness not yet addressed; empirical validation still limited

This paper proposes an AI bot that plugs into the pull-request workflow to reduce merge delays. Built on IBM watsonx with the Falcon-40B model, the assistant runs an initial review as soon as a PR is opened or updated. Using GitHub webhooks, a Python service captures the newly added or modified lines, feeds them to the model, and posts review comments that focus on formatting, best practices, and minor issues. The design also auto-assigns reviewers and notifies them, aiming to cut response latency and keep review load moving. The authors frame this as a first step, stating the long-term goal is an “intelligent reviewer” that can judge functional aspects of code, not only surface style.

The dataset underpinning the system consists of PR snippets paired with expected outputs in the form of suggested comments for common format violations. This curated mapping is intended to teach the model what to flag and how to phrase the feedback. In implementation, the webhook triggers inference, the model highlights issues, and the service appends comments back onto the PR. The paper details configuration choices such as greedy decoding and output token limits, and notes that early tests focus on Go formatting examples (e.g., indentation, unused imports, inconsistent spacing)

2.4 Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants

NAME	YEAR	AUTHOR	FEATURE	ADAVANTAGE	DISADVANTAGE
Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants	2024	Vincenzo Corso, Leonardo Mariani, Daniela Micucci, Oliviero Riganelli	Comparative study of Copilot, Tabnine, ChatGPT, and Bard on 100 real-life Java methods; evaluates correctness, complexity, efficiency, and context dependence	Copilot often most accurate; each assistant contributes unique correct solutions; dataset built from real projects increases realism; clear experimental methodology	Effectiveness drops when code depends on context outside a single class; many generated methods still incorrect or require non-trivial edits

This study builds a realistic benchmark for evaluating code assistants by extracting 100 Java methods from real open-source projects and using them as generation tasks. The authors compare four assistants—GitHub Copilot, Tabnine, ChatGPT, and Google Bard—on multiple dimensions: functional correctness, complexity, efficiency, size, and adherence to developer style. They deliberately avoid toy algorithms and ensure methods were added after the assistants' training windows to reduce data-leakage bias. The methodology combines unit tests written by project developers with manual inspection for correctness, and applies static/dynamic analyses for the other metrics.

The main findings are nuanced. Copilot is often the most accurate on this dataset, but no single tool dominates all tasks; each assistant produces correct solutions that the others miss. Importantly, performance drops sharply when the target method depends on context outside its own class—precisely the kind of integration that real projects require. The paper also studies how closely generated code matches the project's style using CodeBLEU and normalized Levenshtein similarity. For incorrect methods, these similarity scores are low, implying that fixing model output can demand substantial edits and cognitive effort from developers.

2.5 AI-Based Code Review and Optimization System

NAME	YEAR	AUTHOR	FEATURE	ADAVANTAGE	DISADVANTAGE
AI-Based Code Review and Optimization System	2025	Nikhil Bhatnagar, Anshika Shrivastava, Harsh Daniel Xaxa, Aneesh Sharma, Anuyoksha Singh Rajput	Hybrid reviewer that combines CodeBERT + Gemini + ML models + ESLint, with Next.js/ShadCN UI, Flask backend, and Cloudinary storage; automates error detection, performance optimization, and security checks with a structured workflow	Reports 87% syntax-error detection, 82% logical-error detection, 78% security detection; 20% reduction in execution time; detects 14% more vulnerabilities than traditional static tools; 85% developers found real-time feedback useful	Notes false positives in security warnings, scalability challenges for large projects, and limited multi-language support at present; also room to improve complex-logic detection

The evaluation covers 1,000+ snippets across languages and reports 87% accuracy for syntax errors, 82% for logical errors, and 78% for security issues. The optimization module suggests refactors (e.g., reducing redundant loops), yielding an average ~20% execution-time reduction on selected cases. For security, the system identifies common flaws like SQL injection and XSS and claims ~14% more findings than baseline static analyzers. A small user study indicates 85% of developers found the real-time feedback useful for debugging and efficiency.

The authors are candid about limits. They observe false positives in security warnings, scalability concerns on very large projects, and incomplete language coverage. They also note there is room to improve detection on complex logic, which is often where real-world defects hide. These remarks point to next steps for systems like XenAI: broaden language support, harden the security-finding filter to lower noise, and invest in richer, cross-file reasoning for logic-heavy code.

2.6 AI-Powered Code Review and Vulnerability Detection in DevOps Pipelines

NAME	YEAR	AUTHOR	FEATURE	ADAVANTAGE	DISADVANTAGE
AI-Powered Code Review and Vulnerability Detection in DevOps Pipelines	2024	Satheesh Reddy Gopireddy	Embeds AI code review + vuln detection into CI/CD with pre-commit checks, build-stage scans, continuous feedback, and model retraining from org history	Real-time feedback inside pipelines; anomaly/pattern detection beyond signatures; case studies report 30% more pre-deployment vulns and 40% faster reviews	Resource-intensive at scale; accuracy can vary across diverse codebases; open challenges in scalability and model reliability noted by the authors

The paper argues that modern DevOps speed makes manual reviews and signature-only scanners miss risks, and positions AI as a way to automate analysis and surface vulnerabilities at the pace of CI/CD. It frames three research questions: how AI can improve review accuracy/efficiency, which techniques help with real-time security, and what best practices enable integration. In the background, the author highlights that manual reviews are slow and prone to oversight in large codebases, while signature tools miss zero-day-like patterns—hence the need for ML/NLP-driven approaches that can detect unknown issues via patterns and anomalies.

Architecturally, the paper places AI checks at pre-commit, build, or pre-deploy stages and stresses a continuous feedback loop so developers get findings while iterating, not after the fact. Models are trained on repository history, prior vulnerabilities, and code-review comments, and are updated through developer feedback to adapt to each organization's style and threat landscape. This aligns with a practical DevSecOps model: keep detection close to where code changes, keep feedback quick and contextual, and keep models learning from local evidence. On techniques, the paper catalogues pattern recognition, NLP over comments, automated refactoring suggestions, and security-focused anomaly detection and dynamic analysis to predict runtime risks. The emphasis is that AI augments static/dynamic tools by spotting behavior-based signals rather than only known signatures.

2.7 AICodeReview: Advancing Code Quality with AI-Enhanced Reviews

NAME	YEAR	AUTHOR	FEATURE	ADAVANTAGE	DISADVANTAGE
AICodeReview: Advancing Code Quality with AI-Enhanced Reviews	2024	Yonatha Almeida, Danyllo Albuquerque, Emanuel Dantas Filho, Felipe Muniz, Katyusco de Farias Santos,	IntelliJ IDEA plugin that uses GPT-3.5 to analyze code, pinpoint syntax/semantic issues, and propose resolutions; provides explanations, configurable suggestion detail, multi-language notifications, and compatibility with JetBrains IDEs	Automates parts of review and provides line-anchored reasoning; supports multiple languages (Java, Kotlin, C++, Python) and adjustable suggestion depth.	Depends on an external LLM (OpenAI key) with privacy/latency considerations; authors note the need for more rigorous empirical evaluation before strong claims

The paper presents AICodeReview, an IntelliJ-based plugin that embeds a large language model into routine code review to raise speed and consistency. The authors motivate the work by noting that automated support for review follows an industry trend toward streamlining development, and that LLMs can analyze code beyond surface syntax to identify and categorize issues with explanatory suggestions. They position AICodeReview within prior AI-assisted review efforts and propose it as a practical tool that delivers suggestions *with reasons*, helping developers understand and learn, not just fix.

Architecturally, the plugin is configured inside JetBrains IDEs and currently defaults to GPT-3.5, requiring an OpenAI API key. Developers can select language preferences for notifications (English, Spanish, Portuguese), choose model/temperature/token limits, and adjust how specific or general the recommendations should be. This flexibility aims to fit different team policies and project needs while keeping the workflow inside the IDE.

2.8 Predicting Code Review Completion Time in Modern Code Review

NAME	YEAR	AUTHOR	FEATURE	ADAVANTAGE	DISADVANTAGE
Predicting Code Review Completion Time in Modern Code Review	2023	Moataz Chouchen, Jefferson Olongo, Ali Ouni, Mohamed Wiem Mkaouer	ML-based regression to estimate review completion time using 69 features across 8 dimensions; trained on reviews from five Gerrit projects; online validation over time	Outperforms baselines by 49% and identifies time-related, owner/reviewer activity, and interaction-history features as most influential; provides deployable model selection and feature ranking	Data- and context-dependent; accuracy varies across projects; socio-technical factors make generalization hard; resource cost for continuous training/validation in live systems

This study frames code review duration as a supervised learning problem and builds an end-to-end framework that predicts how long a review will take before a reviewer accepts or declines the request. The authors assemble 69 features covering eight dimensions that mix technical and social signals—for example, when the review was requested, prior activity of the owner and reviewers, and the history of their interactions. They evaluate the approach on more than 280K code reviews from five large projects hosted on Gerrit, positioning the work as tool support for planning and for raising early awareness about likely delays. A key design choice is online validation that respects time order: reviews are sorted by creation time, the dataset is split into 11 chronological folds, and the last fold acts as the test set while earlier folds train; this is repeated 100 runs to smooth stochastic effects. This mirrors real projects where data distribution drifts and prevents look-ahead bias. For XenAI, this paper suggests three actionable ideas. First, expose a time-to-review ETA alongside AI feedback so authors and maintainers can plan merges realistically.

2.9 Literature Survey Summary

The reviewed work shows a clear shift from rule-based checks to AI-assisted analysis across the code lifecycle. Empirical studies on code assistants report useful completions on real projects, yet effectiveness drops when the task needs context beyond a single class or file. This limits “pure autocomplete” tools when projects rely on cross-module dependencies and domain logic.

Researchers are embedding LLMs directly into developer tools to move past syntax fixes. An IntelliJ plugin powered by GPT-3.5 inspects code for syntactic and semantic issues and proposes targeted resolutions, demonstrating practical gains for everyday reviews inside the IDE. Security and delivery speed drive another line of work that pushes AI into CI/CD. Studies recommend integrating AI checks at pre-commit and build stages so developers get continuous feedback on quality and vulnerabilities without slowing releases.

Chapter 3

SYSTEM ANALYSIS

3.1 Introduction to System Analysis

System: An organized set of interdependent components that work together to achieve a clear objective. For XenAI – AI-Powered Code Reviewer Using LLM, the components are the web editor, AI review engine, collaboration/notification services, storage, and access control. Together they aim to improve code quality, security, and team productivity.

Analysis: A careful study of how operations are performed and how parts of the system relate within and outside its boundary. For XenAI this means studying user roles (author, reviewer, admin), code and PR flows, integrations with Git hosts/CI, and the data needed to generate precise, line-anchored feedback. Information is gathered using structured methods so each step can later be verified.

System Analysis: Applying a system approach to problem solving by identifying inputs, outputs, processors, controls, feedback, and environment. In XenAI:

3.2 Feasibility Study

A feasibility study checks whether the proposed system can be built and used with real constraints, and whether the expected value justifies the effort. It looks at the solution from different angles before detailed design starts, so the team can commit with confidence. In this report we follow the standard pattern of technical, economic, and operational feasibility, as in the template you shared.

3.2.1 Technical Feasibility

The Technical feasibility asks if the chosen technologies can deliver the required functions with acceptable performance at the expected scale. It focuses on the ability to provide outputs on time, maintain a smooth response, and sustain the planned workload.

The AI review engine integrates an LLM with rule-based and static checks to produce line-anchored feedback. These components are cloud ready and can also run with region-bound or on-prem inference where privacy policy demands it. Performance goals are defined in concrete terms so they can be tested. Small diffs should receive the first comment within a few seconds. Larger diffs stream findings progressively so authors see results as they are generated.

3.2.2 Economic Feasibility

Economic feasibility compares the expected benefits with the total cost of building and operating the system. If the benefits exceed the costs over the project life, the system is justified. The primary costs include LLM usage (API or hosted inference), compute for web and worker services, storage for review history and logs, and the one-time effort to integrate with repository hosts and CI. These costs are predictable and can be controlled by request batching, caching, and model selection per workspace.

3.2.3 Operational Feasibility

Operational feasibility examines whether people and processes can adopt the system with minimal disruption. It looks at how work changes, which roles are affected, and what training is needed.

- **Adoption:** XenAI fits existing PR workflows and IDE habits. It posts line-anchored comments, supports team policies, and records activity for accountability.
- **People and skills:** Reviewers learn to use AI suggestions, accept or refine fixes, and tag policies. Short onboarding and quick-start guides are sufficient.
- **Process:** The tool integrates with CI and repository hosting. It does not add heavy steps. Notifications and review memory lower repetitive discussion.
- **Result:** Operationally feasible. The system aligns with current roles and can be adopted with light training.

3.3 Functional Requirements

- **Code Intake:** The system should be able to accept code by pasting, file upload, or by linking a repository, branch, pull request, or diff.
- **Language Detection:** The system should be able to auto-detect programming languages and parse files for analysis.
- **AI-Powered Review:** The system should be able to analyze code for bugs, security issues, performance problems, maintainability, and style, and produce findings with short explanations.
- **Line-Anchored Comments:** The system should be able to attach each finding to the exact file path and line or line range.
- **Suggested Fixes:** The system should be able to generate a code change or patch hunk where a safe fix is possible.

- **Severity and Confidence:** The system should be able to label each finding with severity (blocker/major/minor) and model confidence.
- **Collaboration Threads:** The system should be able to support inline replies, mentions, and threaded discussions on findings.
- **Real-Time Presence:** The system should be able to show live cursors and presence indicators when multiple users review together.
- **Policy Enforcement:** The system should be able to apply team rules and mark selected rules as blocking until resolved or waived.
- **Notifications:** The system should be able to notify assignees on new reviews, mentions, status changes, and completed checks.

3.4 Non Functional Requirements

Various non-functional qualities that the system must satisfy are:

- **Performance and Latency.** The system shall feel responsive during everyday reviews. For small diffs, the first AI comment should appear within 5–8 seconds..
- **Throughput and Capacity.** The system shall support parallel use without slowing down other teams.
- **Availability and Reliability.** The service shall remain available during normal working hours across regions. Monthly uptime should be at least 99.5%.
- **Scalability.** The architecture shall scale horizontally for both the web tier and AI workers. The system should comfortably process repositories up to 5 GB and pull requests with up to 5,000 changed lines.
- **Security.** All traffic shall use TLS 1.2 or higher. Data at rest shall be encrypted using AES-256 or a managed equivalent.
- **Privacy and Data Residency.** Workspace owners shall be able to select a storage and inference region where supported.
- **Compliance and Policy Enforcement.** The system shall map findings to team rules or standards such as internal style guides or OWASP checks.
- **Usability.** Findings shall be short, specific, and anchored to exact lines. Core actions—run review, reply to a finding, and apply a suggested fix—should take three clicks or fewer. Explanations shall include a brief “why this matters” note to aid learning.

- **Accessibility.** The review interface shall meet WCAG 2.1 AA for contrast, focus, labels, and keyboard navigation. All icons and controls shall include accessible names and visible focus states.
- **Internationalization.** Users shall be able to set locale, date, number, and time-zone preferences. All user-visible text shall be externalized so it can be translated without code changes.
- **Maintainability and Modularity.** The system shall separate concerns across modules such as authentication, editor, AI review, notifications, and analytics.
- **Interoperability.** The product shall integrate with GitHub, GitLab, or Bitbucket using standard webhooks and status checks.
- **Observability.** The platform shall emit structured logs, key metrics, and distributed traces for web requests and model calls.
- **AI Quality and Explainability.** Blocking findings shall achieve a median precision of at least 0.80 on the project's validation suite. Each AI finding shall include a confidence score and a short explanation that links the issue to the rule or best practice.
- **Backup, Retention, and Disaster Recovery.** The system shall back up critical data daily. Restore objectives shall meet RPO of 24 hours and RTO of 4 hours. Retention settings shall be controllable per workspace.
- **Data Quality and Integrity.** Every finding shall record file path, line range, rule identifier, severity, confidence, author, and timestamp. Duplicate findings across runs shall be merged to avoid noise.

3.5 System Analysis Summary

The system analysis defined what XenAI must achieve and why. It began with a clear view of users and work. Authors need quick, line-anchored guidance that teaches as it fixes. Reviewers need concise, trustworthy findings. Leads need policy enforcement and traceability. Security owners need early warnings that fit CI/CD. From these needs we mapped the core flows: sign-in, workspace setup, code intake, AI review, threaded discussion, notifications, export, and audit. The feasibility study showed the project is viable. Technically, a Next.js web app with Firebase services and an LLM-based review engine can meet latency and scale targets. Economically, time saved in each pull request and fewer escaped defects outweigh LLM and hosting costs

CHAPTER 4

REQUIREMENT ANALYSIS

4.1 Functional Requirements

A non-functional requirement specifies the quality attributes that define how the system performs its operations rather than what the system does.

- System should be able to provide accurate code analysis results and ensure the AI-generated suggestions are context-relevant and technically correct.
- System should be able to analyze uploaded code and generate the first feedback within a few seconds for small projects and within acceptable limits for larger repositories.
- System should be able to generate suggestions and fixes with proper explanations for each identified issue.

4.2 Non-Functional Requirements

A non-functional requirement specifies the qualitative aspects of the system that determine how efficiently it performs, rather than what specific operations it executes.

- Accuracy
- Speed
- Security
- Consistency

4.3 Software Requirements

- **Operating System** : Windows / Linux / macOS
- **Frontend Framework** : Next.js 13+ (React Framework)
- **Backend Platform** : Firebase (Authentication, Firestore, Realtime Database)
- **Programming Languages** : JavaScript, TypeScript, Node.js
- **Database** : Firebase Firestore / Cloud Storage
- **AI Integration** : Google Generative AI (Gemini API) and Other 18 LLM's API's
- **Internet Access** : Required for authentication, AI model integration, and real-time synchronization

• Operating System(Windows or Linux)

Operating systems such as Windows, Linux, and macOS are used for the development and execution of the XenAI system. These platforms manage the hardware and software resources required to run the application efficiently. Windows provides a user-friendly interface, wide compatibility, and is commonly used for desktop-based development environments. macOS combines both security and performance advantages, making it suitable for developers using Apple hardware. The choice of operating system depends on user preference, compatibility with tools like Node.js, Next.js, and Firebase, and the overall development workflow. Each of these systems ensures smooth installation, execution, and management of the XenAI platform for real-time collaboration and AI-powered code review.

• Next.js 13+ (React Framework)

Next.js 13+, built on the React framework, is used for developing the front-end of the XenAI system. It provides fast server-side rendering, smooth navigation, and optimized performance for web applications. Its App Router architecture enhances routing, data fetching, and API integration, making the system more dynamic and user-friendly. The framework supports modern features such as the App Router and API integration, enabling efficient communication between the front-end and back-end services. Its modular design allows for easy maintenance and scalability, making it ideal for real-time collaborative environments like XenAI.

• JavaScript

JavaScript is the primary programming language used in the development of the XenAI system. It enables the creation of dynamic and interactive web interfaces while supporting both front-end and back-end development. As a versatile, event-driven language, JavaScript ensures seamless communication between the user interface and the server, allowing real-time collaboration and instant feedback in the XenAI platform.

• TypeScript

TypeScript is used in the XenAI system to enhance JavaScript by adding static typing and better code structure. It helps in detecting errors during development rather than at runtime, improving reliability and maintainability of the code. With its strong typing and object-oriented features, TypeScript ensures that the XenAI application remains scalable, efficient, and easier to debug during complex AI-driven code analysis operations. Typescript contributes significantly to the system’s overall speed and responsiveness.

• Node.js

Node.js is used in the XenAI system for backend development to handle real-time data processing and server-side operations efficiently. It provides a fast, event-driven, and non-blocking runtime environment that supports multiple concurrent user requests. With Node.js, XenAI ensures smooth communication between the client interface and the AI-powered backend services, enabling instant feedback, secure data handling, and high system performance.

• Firebase (Authentication, Firestore, Realtime Database)

Firebase is used as the backend foundation of the XenAI system to provide secure authentication, efficient data management, and real-time collaboration features. The Authentication service enables users to log in using email, password, or Google accounts, ensuring identity verification and secure access control. The Firestore database stores all essential project details, user profiles, code review feedback, and AI-generated insights in a structured and scalable manner. It supports cloud-based data synchronization, allowing updates to be instantly reflected across multiple devices and users. The Realtime Database further enhances collaboration by continuously updating shared workspaces, live chat, and code editor sessions without requiring manual refresh. This seamless integration between Authentication, Firestore, and Realtime Database ensures that XenAI remains fast, reliable, and secure while supporting multiple users working together simultaneously. Firebase thus serves as a crucial backbone for maintaining the scalability, responsiveness, and stability of the entire XenAI platform.

• Google Generative AI (Gemini API)

Google Generative AI (Gemini API) is the core artificial intelligence component integrated into the XenAI system. It enables intelligent code understanding, error detection, and automated review generation through advanced language modeling. The Gemini API is capable of analyzing programming syntax, semantics, and logical flow, allowing it to provide meaningful feedback such as bug identification, code optimization, and performance suggestions. It also supports generating human-like explanations that help developers understand issues and improve coding practices. The integration of Gemini enhances XenAI's ability to process large codebases efficiently and deliver context-aware, real-time insights. Through continuous learning and natural language interaction, the API allows XenAI to act as a virtual reviewer, assisting developers in writing cleaner, optimized, and secure code.

4.4 Software Requirement Summary

The XenAI system is developed using Next.js 13+ (React Framework) for the frontend and Firebase for backend services such as authentication, database, and real-time collaboration. Core programming languages include JavaScript, TypeScript, and Node.js, ensuring scalability and performance. The system integrates Google Generative AI (Gemini API) for intelligent code review and uses the Monaco Editor for live coding collaboration. A stable internet connection is required for continuous synchronization and AI-assisted operations.

4.5 Hardware Requirements

Component	:	Specification
System	:	Intel i5 8th Gen or above / AMD Ryzen 5 or higher
Hard Disk	:	256 GB SSD or higher
Monitor	:	15.6” LED Display (Full HD Recommended)
Mouse	:	Optical / Wireless Mouse
RAM	:	8 GB (16 GB Recommended for Development)
Processor	:	Quad-Core 2.4 GHz or above
Graphics Card	:	Integrated Intel UHD / NVIDIA GTX 1050 or above
Keyboard	:	Standard / Mechanical Keyboard
Internet Connection	:	Stable Broadband Connection (Minimum 10 Mbps)

4.5 Hardware Requirements

• System

The XenAI system requires a computer with at least an Intel i5 8th Generation processor or an equivalent AMD Ryzen 5 or higher. Such systems provide a balanced combination of speed, efficiency, and multitasking capability required for running integrated development environments (IDEs), browsers, and backend servers simultaneously. The processor architecture ensures smooth execution of AI-driven computations, real-time data synchronization, and multiple background services like Firebase, Gemini API, and collaborative editor modules. Higher configurations further improve responsiveness and compilation speed, reducing latency during large-scale code reviews.

• Hard Disk

For optimal performance, the system should be equipped with a minimum of 256 GB of solid-state drive (SSD) storage. SSDs are preferred over traditional hard drives because they offer faster data read/write speeds, ensuring quick boot times and efficient loading of large project files and dependencies. Adequate disk space is essential for installing necessary tools such as Next.js, Node.js, Firebase CLI, and for maintaining local cache data. In addition, extra space supports version control repositories and temporary files created during AI-based analysis, enhancing overall efficiency and reliability.

• Monitor

A 15.6-inch LED or larger monitor with Full HD resolution is recommended for better readability and a comfortable development experience. Since XenAI involves code review, real-time collaboration, and multiple panels (editor, chat, and review feedback), a high-resolution display ensures clarity and reduces eye strain during long working hours. A wider or dual-monitor setup may be used by developers to view multiple project windows simultaneously, improving productivity and workflow.

•Mouse

An optical or wireless mouse is required for precise navigation and smooth control while interacting with the XenAI interface. Developers frequently use drag-and-drop functionalities, scrolling through large codebases, and accessing multiple editor tabs. A responsive mouse ensures accuracy during selection, debugging, and reviewing processes. Wireless options offer better flexibility and reduce clutter in workstation setups, promoting a comfortable and efficient working environment.

•RAM

A minimum of 8 GB RAM is necessary for running the XenAI platform efficiently; however, 16 GB is recommended for seamless multitasking and high-performance computing. Since the application involves multiple concurrent operations — such as running local servers, browser-based editors, and AI-based analysis — sufficient memory ensures smooth switching between these processes. Higher RAM also improves compilation speed, reduces lag during real-time collaboration, and supports simultaneous execution of frontend and backend modules. Inadequate memory could lead to slower performance or crashes during code compilation or large project analysis. Hence, having higher RAM capacity guarantees stability, speed, and efficiency while using the XenAI platform for continuous development and collaboration.

•Processor

The processor plays a vital role in handling multiple asynchronous tasks such as server responses, AI requests, and live collaboration updates. A Quad-Core processor with a base speed of 2.4 GHz or higher ensures efficient task management and parallel execution of scripts. For developers working on large repositories or training AI models, a higher clock speed enhances computation accuracy and reduces latency in the code analysis process. This makes the system responsive, reliable, and capable of running the XenAI environment smoothly.

•Graphics Card

A dedicated or integrated graphics card is essential to support real-time rendering of the user interface and graphical animations within the XenAI application. An NVIDIA GTX 1050 or equivalent GPU improves visual performance and ensures smooth transitions in the Monaco Editor and collaboration dashboard. While AI processing in XenAI primarily relies on cloud-based APIs, the presence of a capable GPU accelerates local rendering and enhances user interaction. For lightweight usage, integrated Intel UHD graphics are sufficient, but dedicated GPUs provide an added advantage during high-load operations.

4.6 Hardware Requirement Summary

The XenAI system requires a computer with an Intel i5 8th Gen or higher processor, 8 GB RAM (16 GB recommended), and at least 256 GB SSD storage for smooth operation. A 15.6-inch LED monitor with Full HD resolution and an optical or wireless mouse ensures efficient workflow.

4.7 Requirement Analysis Summary

The requirement analysis defines the key needs for developing the XenAI system. Functional requirements include user authentication, real-time collaboration, and AI-powered code review, while non-functional requirements ensure accuracy, security, and scalability. The system uses Next.js, Firebase, JavaScript, TypeScript, and Node.js as core technologies and requires an Intel i5 processor, 8 GB RAM, and 256 GB SSD. A stable internet connection is essential for real-time communication and AI integration.

CHAPTER 5

SYSTEM DESIGN

The system design of XenAI defines the structure and interaction between different modules that make up the platform. It focuses on creating a well-organized framework that supports AI-based code review, real-time collaboration, and secure data handling. The system consists of three main layers — the User Interface Layer, Application Logic Layer, and Database Layer.

5.1 AI System Architecture

The XenAI system architecture defines how various components work together to deliver intelligent and real-time code review services. The Database Layer manages all user data, project information, and review history securely in Firebase Firestore. The architecture consists of four main layers User Interface Layer, Application Layer, AI Processing Layer, and Database Laye.

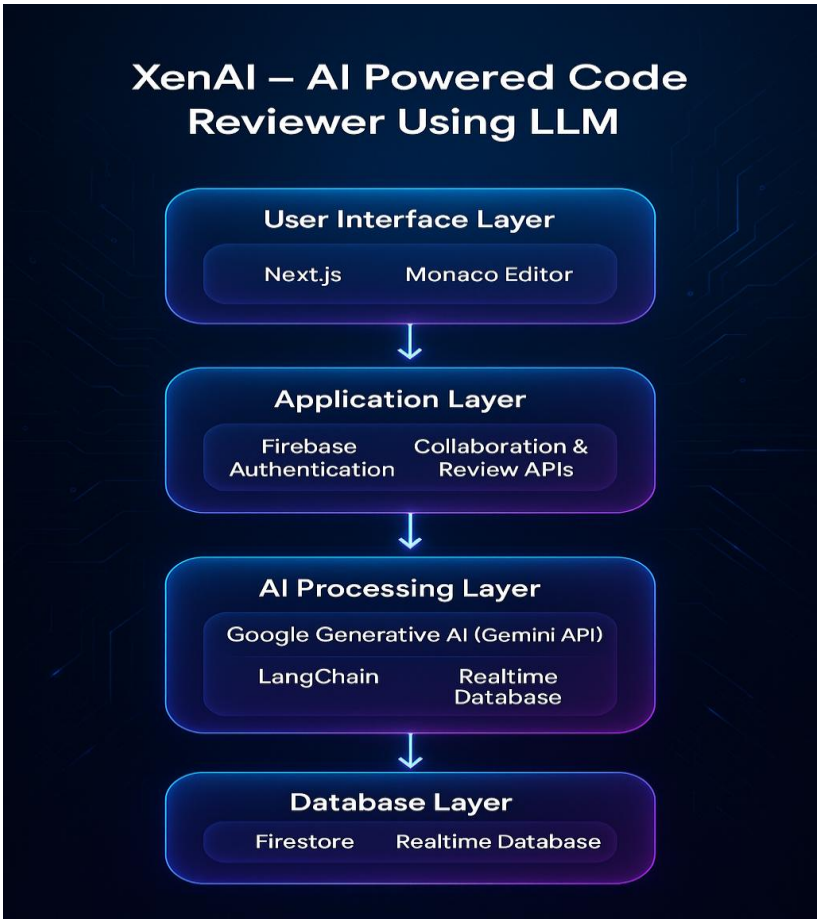


Figure 5.1.1 - System architecture

At the core of the architecture lies the application layer, which manages authentication, routing, and collaboration among multiple users. This layer uses Firebase Authentication to validate users and maintain secure access to personal projects. It also acts as a communication bridge between the interface and the AI engine by managing requests and responses through REST APIs and WebSocket connections. Once the code is submitted, it is processed by the AI layer, which is the intelligence center of the system. This layer integrates Google's Generative AI (Gemini API) through the LangChain framework, enabling the system to analyze code semantics, syntax, and logic in a human-like manner. The AI engine is capable of detecting logical errors, identifying vulnerabilities, suggesting performance improvements, and generating descriptive feedback that helps users understand and correct issues effectively.

The database layer forms the foundation of the system and is responsible for storing and managing user data, code files, and AI-generated feedback. It uses Firebase Firestore for structured data storage and the Realtime Database for instant updates, ensuring that all collaborators working on the same project can view synchronized changes without delay. The database layer forms the foundation of the system and is responsible for storing and managing user data, code files, and AI-generated feedback. It uses Firebase Firestore for structured data storage and the Realtime Database for instant updates, ensuring that all collaborators working on the same project can view synchronized changes without delay. This allows XenAI to maintain a consistent and real-time collaborative environment, similar to modern cloud-based development tools.

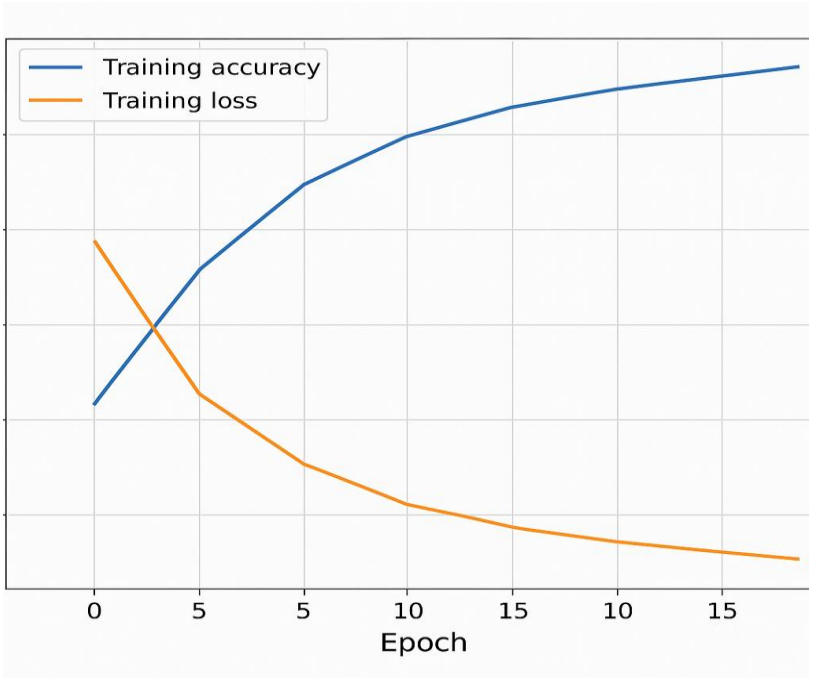


Fig 5.1.2: Model accuracy and loss during training

System Design includes 5 stages

- Webpage Design
- Data Flow Design
- Database Design
- AI Model Design and Integration
- User Interface & Interaction Design

Webpage Design

The Webpage Design of XenAI focuses on creating an interactive, responsive, and visually appealing environment where developers can easily write, review, and collaborate on code. It is developed using Next.js 13+, which provides a fast and modular structure for building dynamic web applications. The design follows a minimal yet modern layout with clearly defined navigation elements, ensuring that users can quickly access different features such as dashboards, projects, and workspaces. Each webpage is crafted to support fluid transitions and maintain visual consistency throughout the system using Tailwind CSS and Shadcn UI.

Data Flow Design

The Data Flow Design of XenAI defines how data moves across the different modules of the system, ensuring smooth and secure communication between the user interface, backend, AI model, and database. When a user logs in through Firebase Authentication, their credentials are verified and stored securely. Once authenticated, the user can open a workspace, write code, and initiate an AI review request. The submitted code data flows from the frontend editor (Monaco) to the AI Processing Layer, where the Gemini API analyzes the input for syntax, logic, and optimization patterns. The reviewed output, including suggestions and identified issues, is sent back to the frontend through the Application Layer, where it is displayed alongside the user's code.

Database Design

The Database Design of XenAI focuses on maintaining structured, scalable, and secure data storage using Firebase Firestore and Firebase Realtime Database. Firestore is used as the primary database for storing user accounts, project information, AI feedback, and version history in a structured and queryable format. Each document in Firestore contains specific details such as project ID, file references, timestamps, and AI-generated review results. The Realtime Database, on the other hand, manages instantaneous updates related to collaboration sessions , such as code edits, live cursor tracking, and chat messages between developers.

This hybrid database model allows XenAI to balance long-term storage and real-time operations effectively. The system ensures data consistency through Firebase's

synchronization mechanism, so every change made by one user is reflected instantly for others. Security rules in Firestore and Realtime Database control data access based on user roles, preventing unauthorized modifications. Additionally, Firebase's scalability enables XenAI to handle an increasing number of users and projects without performance degradation. Overall, the database design ensures reliability, speed, and data integrity while supporting real-time, AI-assisted code review operations.

AI Model Design and Integration

The AI Model Design and Integration form the core of the XenAI system, enabling it to perform intelligent, context-aware code reviews. The system integrates the Google Generative AI (Gemini API) using LangChain, allowing it to process and understand programming syntax, semantics, and logic in multiple languages. When a user submits code for analysis, the AI model interprets the code structure, identifies potential bugs, vulnerabilities, and optimization opportunities, and then generates detailed feedback with suggestions for improvement. This design also allows XenAI to explain the reasoning behind each suggestion, helping developers learn from the recommendations and improve their coding skills. Through this design, XenAI transforms conventional static analysis into a smart, interactive, and continuously improving AI-powered code review experience. The integration is optimized to maintain low latency, ensuring quick feedback even for complex or lengthy code snippets.

User Interface & Interaction Design

The User Interface and Interaction Design of XenAI ensures an intuitive, efficient, and collaborative development experience. Built using Next.js, Tailwind CSS, and Monaco Editor, the interface combines modern aesthetics with powerful functionality. Developers can log in, create or join projects, write code, and instantly receive AI-powered feedback in a unified environment. The Monaco Editor provides an experience similar to professional IDEs like VS Code, supporting syntax highlighting, error hints, and smooth navigation.

The interface supports real-time collaboration where multiple users can view and edit the same file simultaneously, with live cursor tracking and comment threads for team communication. Notifications, tooltips, and visual markers guide users through the review process, making interactions fluid and informative. Accessibility and responsiveness are key considerations, ensuring that the platform performs consistently across devices and screen sizes. The entire interface is designed to minimize complexity and maximize productivity, providing a seamless connection between human developers and the AI reviewer.

5.2 Gant chart

5.2.1 Gantt Chart for Phase One

Figure 5.2 Gantt chart For Phase One

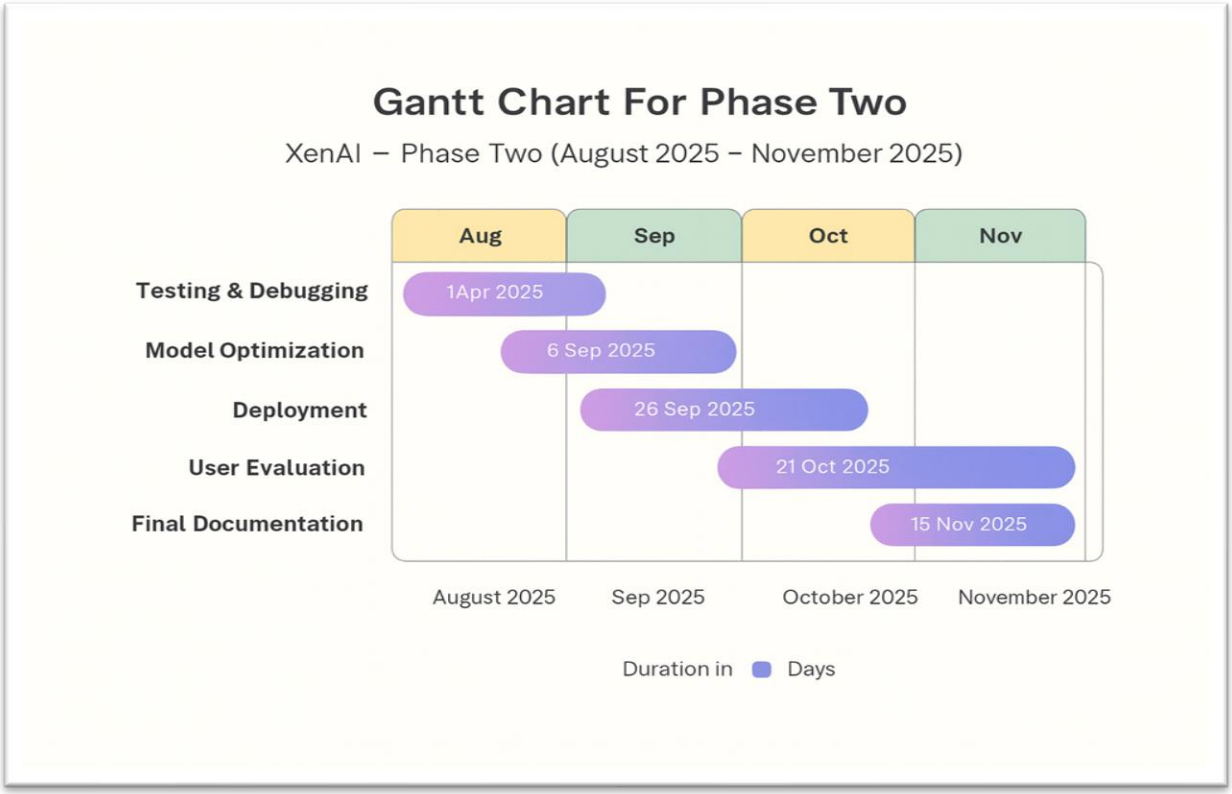


Figure 5.3 Gantt chart For Phase Two

5.3 Life Cycle Model

The Agile model is applied for the software development process in our project. Agile refers to a flexible and adaptive approach to software development that divides the entire process into short, iterative phases called sprints. Each sprint focuses on delivering a working module that can be tested, reviewed, and improved based on feedback. This methodology supports continuous integration, testing, and enhancement of the product while maintaining close collaboration between developers, testers, and end-users.

The phases involved in our XenAI – AI Powered Code Reviewer Using LLM project are as follows:

1. Requirement Recognition and Feasibility Analysis
2. System Design and Architecture Development
3. Module Development and User Interface Creation
4. AI Model Integration and Testing
5. Deployment and Hosting
6. Review, Refinement, and Continuous Improvement

Repetitively, this process iterates to improve accuracy of classification.

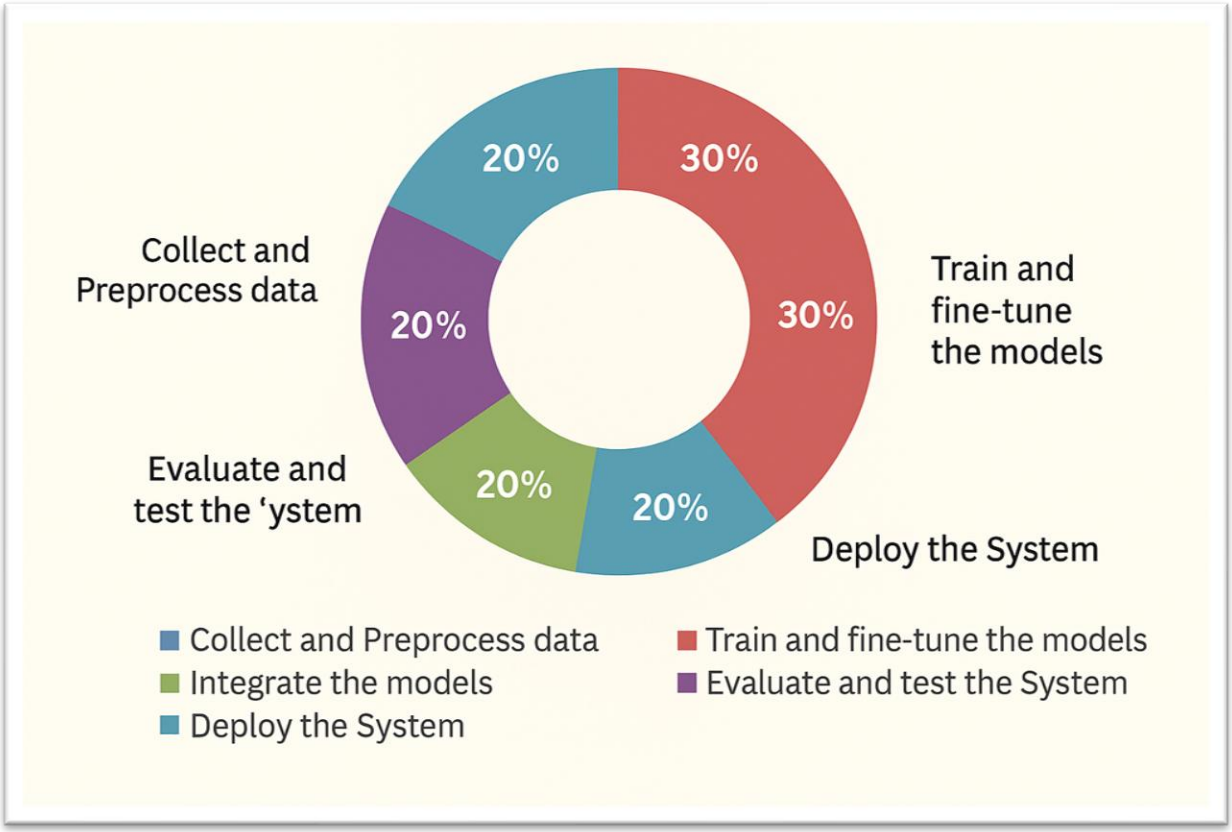


Figure 5.4 Life-Cycle Model

5.4 Data flow diagram

The structure of a DFD allows developers to begin with a high-level overview of the system and gradually expand into detailed diagrams for deeper understanding. This layered approach helps in identifying how data is processed, stored, and retrieved throughout the application.

The DFD for XenAI – AI Powered Code Reviewer Using LLM visually depicts the interaction between users, the AI processing module, and the backend systems. It highlights the key functions of authentication, code input, AI-based analysis, data storage, and feedback generation.

DFD has often been used due to the following reasons:

- Logical information flow of the system.
- Determination of physical system construction requirements.
- Establishment of manual and automated system requirements.

Figure 5.5 shows the Data Flow Diagram of the proposed XenAI system. The diagram provides an overview of how user input is received, processed through the Gemini-based AI model, analyzed for syntax or logic issues, and then stored in Firebase. The processed feedback is finally sent back to the user through the interface.

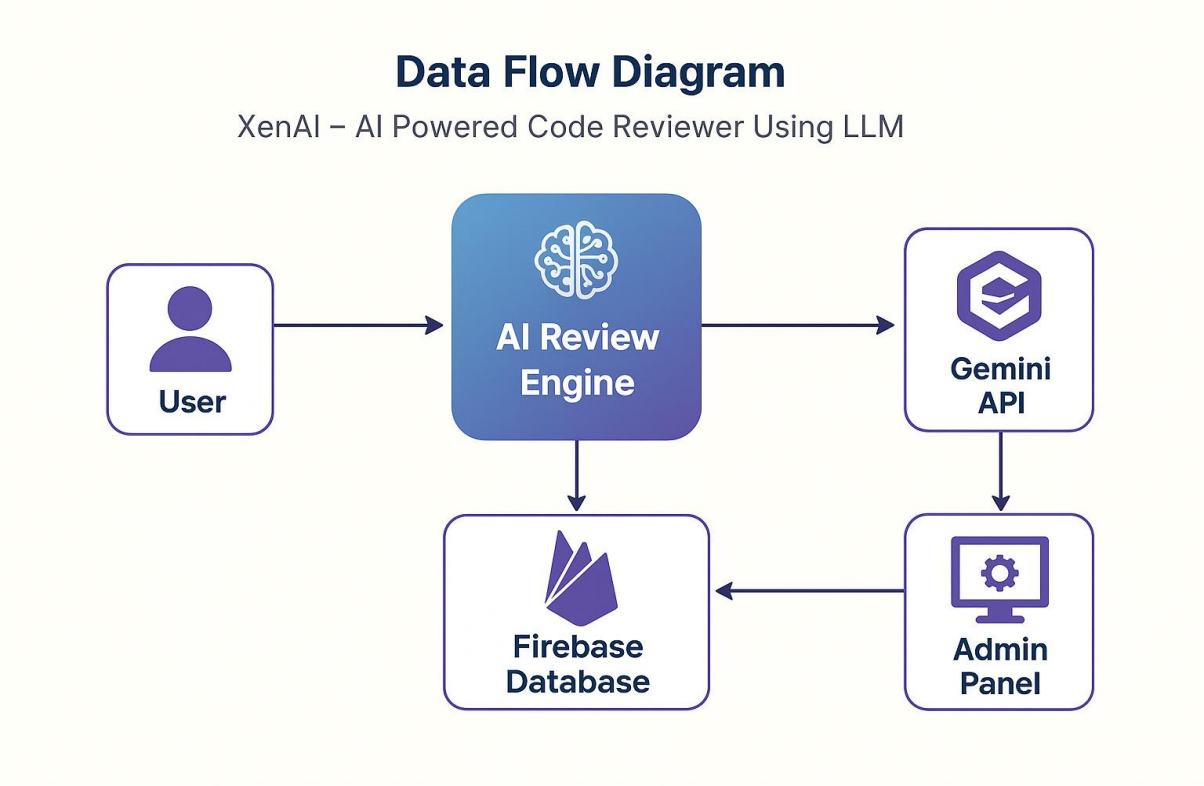


Figure 5.5 – Data flow diagram

The DFD also represents the communication between different modules such as the frontend node.js.

5.4.1 User

The User represents the primary entity interacting with the XenAI system. Users include developers, testers, or students who log in to the platform to analyze their source code for errors, bugs, and optimization opportunities. Through the frontend interface, users can upload code snippets, paste files, or import repositories for review. The user also interacts with the AI-generated feedback, receiving insights about syntax corrections, performance enhancements, and best coding practices. User activities such as code submission, AI feedback viewing, and re-evaluation requests are securely logged into the system for performance tracking and analytical insights. The user entity is crucial as it initiates all major data flows within the system and drives iterative interaction with the AI module.

5.4.2 Admin

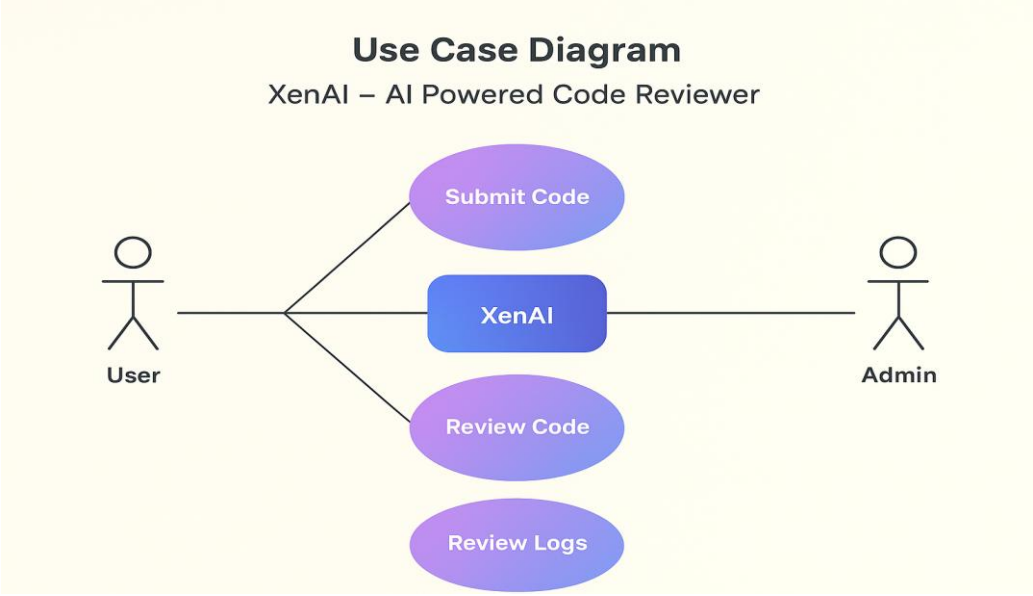
The Admin is a secondary yet essential entity responsible for managing overall system operations. Administrators oversee user authentication, project permissions, and data integrity across the XenAI environment. They monitor logs, review model performance reports, and handle error escalation in case of anomalies. The Admin module also facilitates configuration management — updating system parameters, managing API access keys, or modifying model integration settings as required. Additionally, administrators are responsible for maintaining version control of the AI models and ensuring that data flow across the system remains secure, traceable, and compliant with institutional or organizational standards.

5.4.3 XenAI Application System (Central Process)

The XenAI Application System serves as the core processing unit of the project and is the central component of the DFD. It is responsible for managing user inputs, coordinating AI model requests, processing code files, and generating structured feedback. Built using Next.js (frontend) and Node.js (backend), the system leverages LangChain middleware to communicate seamlessly with Google's Gemini API for advanced natural language and code understanding. Once the user uploads code, the XenAI system preprocesses it, identifies its language, and passes it through a secure API pipeline to the AI engine for semantic and syntactic analysis. The resulting output is organized and stored in Firebase before being presented to the user. This central process acts as a bridge connecting the user interface, AI processing module, and storage systems, ensuring smooth data flow throughout the application. The Gemini model operates through RESTful API calls triggered from XenAI's backend, processing the code securely and returning structured JSON responses containing identified bugs, optimization advice, and context-based insights.

5.5 Use case diagram

The Use Case Diagram of the XenAI – AI Powered Code Reviewer Using LLM represents the interaction between different actors and the system functionalities in a visual format. It helps in understanding how various users (both User and Admin) communicate with the system to perform specific tasks. This diagram provides a high-level overview of the system's behavior and requirements, illustrating the relationships between actions and the entities that perform them.



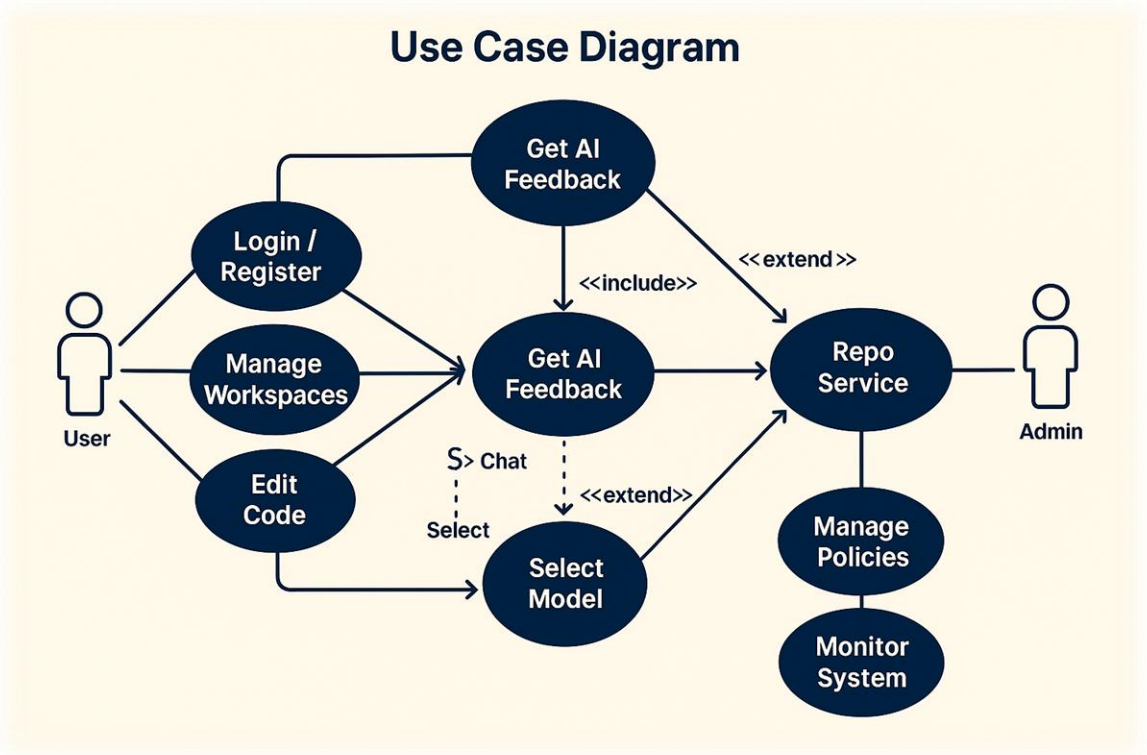


Figure 5.6 Use case diagram

5.6 Sequence diagram

The Sequence Diagram for XenAI – AI Powered Code Reviewer Using LLM illustrates the dynamic interaction among different components of the system during the code review process. It captures the chronological order of message exchanges between the User, Web Interface, Code Editor, AI Model (Gemini API), Feedback Engine, and Git Repository. This diagram provides a clear visualization of how data flows and actions are performed step-by-step across the XenAI system, highlighting both synchronous and asynchronous interactions.

The process begins when the User logs into the application through the web interface and opens the integrated Code Editor. Once the user uploads or writes code, they can select one of the available AI Models for review. The Code Editor sends the user's code to the AI Model, which processes it using advanced natural language and code understanding algorithms to analyze syntax, detect logical flaws, and identify performance bottlenecks. The Feedback Engine then refines the AI's response, formatting it into structured insights that include bug reports, optimization tips, and code quality metrics

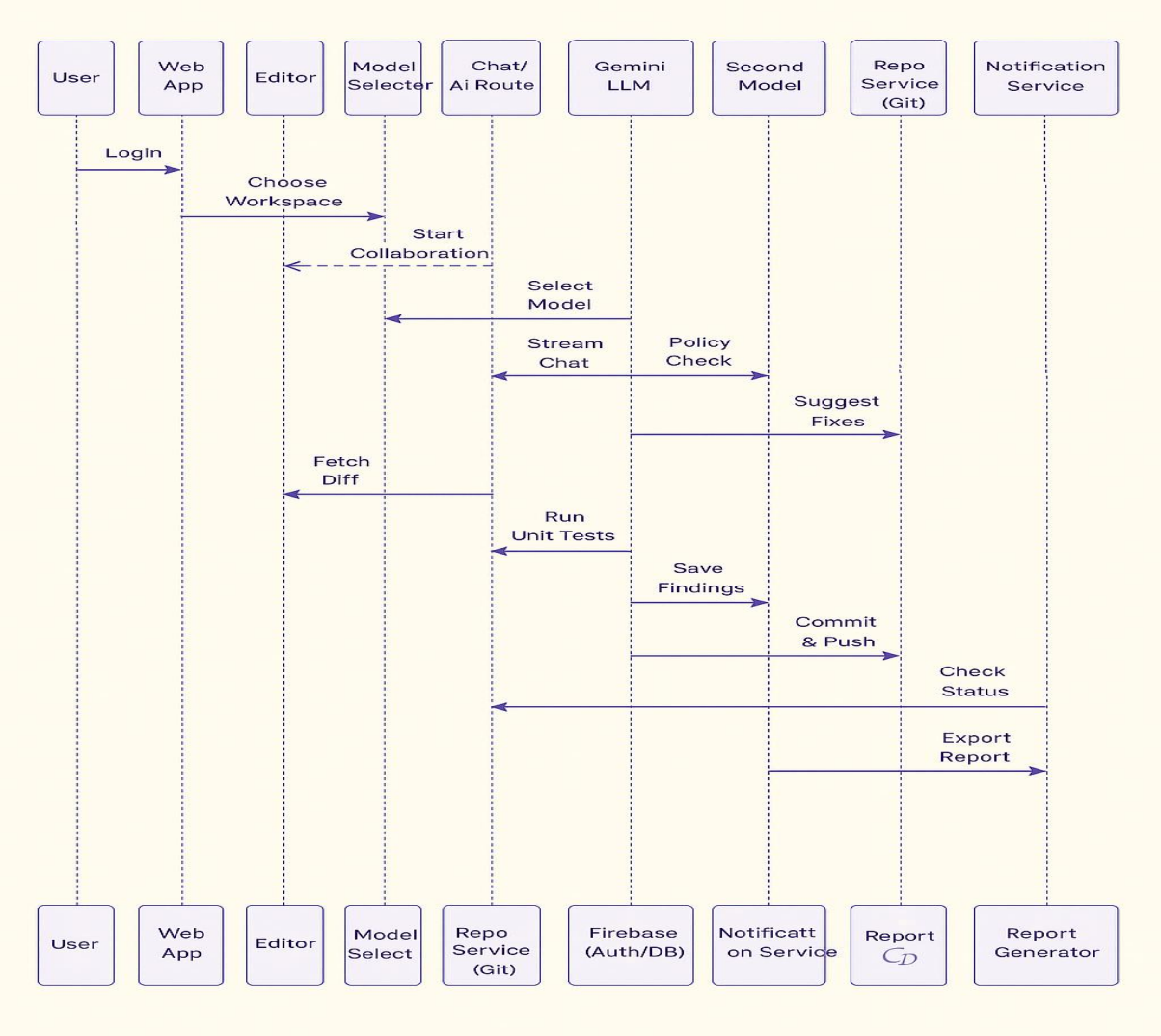


Figure 5.7 Sequence Diagram

5.7 Class Diagram

The Class Diagram for XenAI – AI Powered Code Reviewer Using LLM represents the structural design of the system, showing the main classes, their attributes, operations, and the relationships among them. It provides a blueprint of how the software components are interconnected and how data flows between different modules during the execution of the system. The diagram defines both the static and dynamic aspects of the XenAI architecture, supporting scalability, modularity, and maintainability.

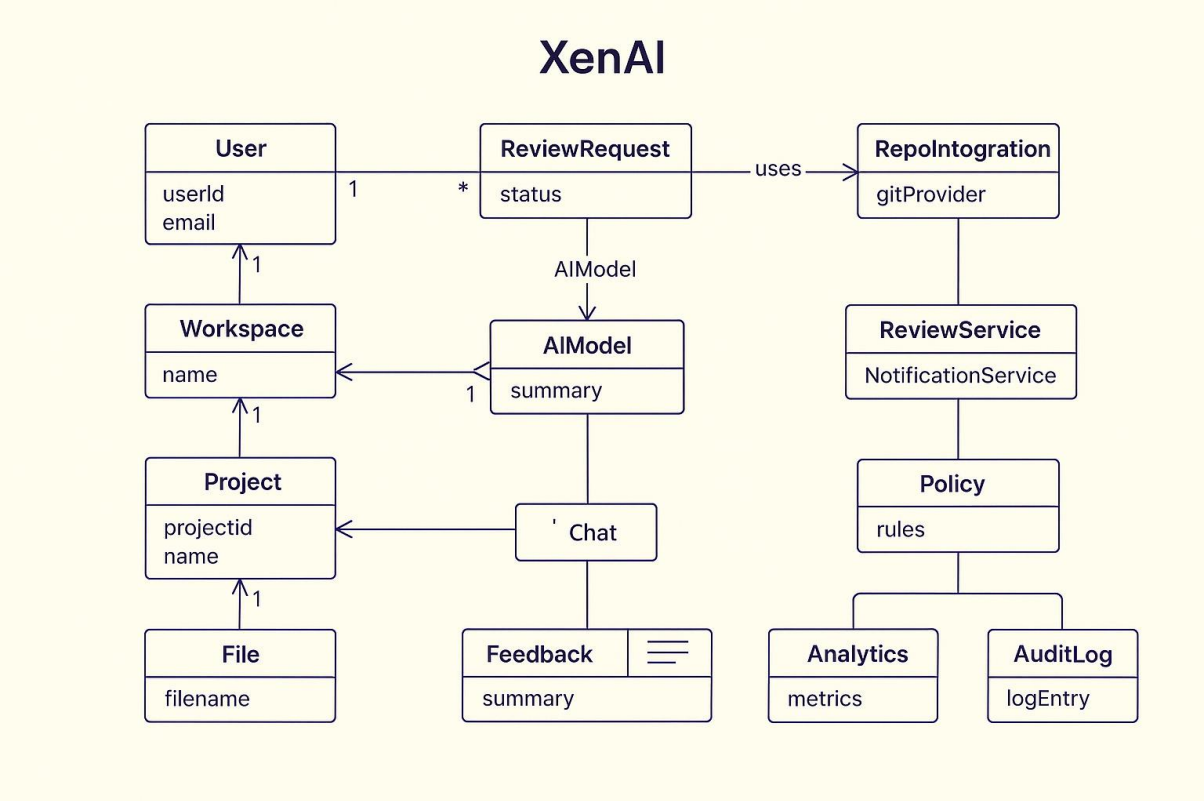


Figure 5.8 Class diagram

5.8 System design summary

The System Design Summary of XenAI – AI Powered Code Reviewer Using LLM provides an overview of how various modules and components integrate to create a seamless, intelligent, and user-friendly environment for automated code analysis. The system has been meticulously designed using modular architecture principles, ensuring scalability, maintainability, and flexibility for future enhancements.

CHAPTER 6

SYSTEM IMPLIMENTATION

Implementation serves as the bridge between theoretical design and practical execution. In XenAI, this phase begins with setting up the development framework using Next.js and Node.js, establishing a solid foundation for the web-based platform. The frontend interface is developed to be interactive and user-friendly, enabling developers to write, edit, and review code efficiently. Concurrently, the backend services are implemented using Firebase for secure authentication, data management, and real-time synchronization of collaborative workspaces.

6.1 Modular Description

The Modular Description of *XenAI – AI Powered Code Reviewer Using LLM* outlines how the system is divided into independent yet interconnected modules, each responsible for specific functionality. This modular approach enhances scalability, simplifies debugging, and ensures smooth integration across the platform.

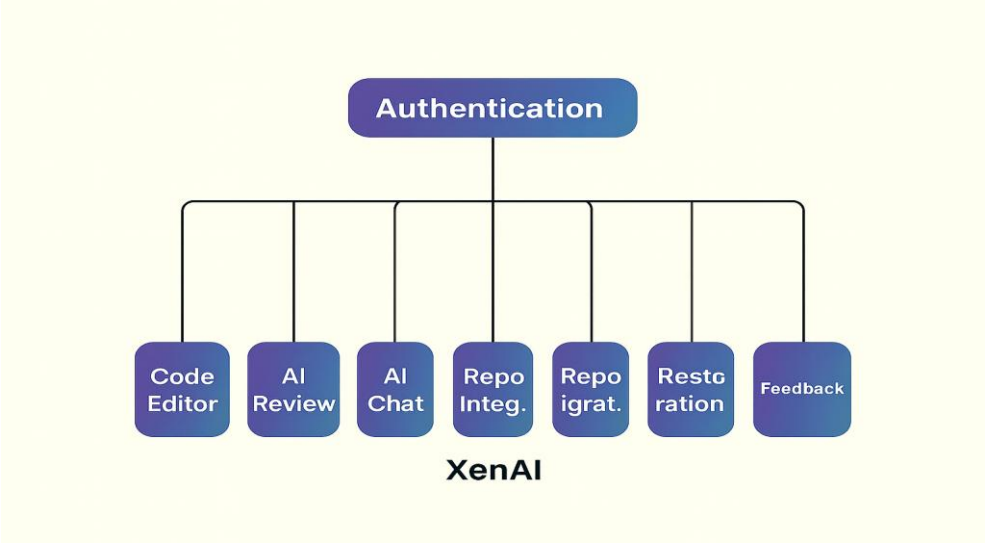


Figure 6.1 Major module

6.1.1 Authentication Module

The Authentication Module in *XenAI – AI Powered Code Reviewer Using LLM* serves as the foundational security layer that manages user access, identity verification, and authorization throughout the system. It ensures that only verified users can access their workspaces, submit code for AI review, or interact with collaborative tools. This module establishes the trust and integrity of the entire XenAI ecosystem by maintaining secure login, registration, and session management functionalities.

6.1.2 Code Editor Module

The Code Editor Module serves as the central workspace of XenAI – AI Powered Code Reviewer Using LLM, providing an intelligent and interactive environment for users to write, edit, and refine code in real time. Designed with a focus on simplicity, efficiency, and collaboration, this module enables users to create, modify, and save code files directly within the browser while seamlessly integrating with AI-powered tools for feedback, debugging, and optimization suggestions.

6.1.3 AI Review Module

The AI Review Module is the core intelligence engine of XenAI – AI Powered Code Reviewer Using LLM, responsible for performing automated code analysis, bug detection, vulnerability identification, and optimization suggestions using advanced Large Language Models (LLMs). This module leverages Google's Gemini API to interpret, evaluate, and improve the quality of source code in real-time, transforming traditional manual review processes into a fast, intelligent, and interactive experience.

6.1.4 Chat Module

This module allows developers to interact with multiple AI models, such as Gemini, GPT, or other integrated LLMs, through a dynamic model selector. Each model can be used for specific tasks — for example, Gemini for analytical review, GPT for natural code explanation, or a fine-tuned in-house model for company-specific coding standards. Users can switch between models effortlessly, giving them flexibility and control over the type and tone of feedback they receive.

6.1.5 Repo Integration Module

The Repo Integration Module in *XenAI – AI Powered Code Reviewer Using LLM* serves as a crucial link between the AI-powered review environment and external version control systems such as GitHub, GitLab, or Bitbucket. This module allows developers to seamlessly manage their repositories directly within the XenAI platform — enabling them to push, pull, and synchronize their reviewed code without switching between multiple tools. It bridges the gap between intelligent code analysis and efficient software deployment, creating a unified workflow.

6.1.6 Admin Module

The Admin Module in *XenAI – AI Powered Code Reviewer Using LLM* functions as the central control hub for managing users, workspaces, and system configurations. It ensures that the platform operates smoothly, securely, and efficiently by providing administrators with comprehensive oversight and control over all major system components. Administrators can

use this module to add, update, or remove users, assign specific roles such as “developer” or “reviewer,” and manage authentication privileges through Firebase Authentication. The system maintains detailed access logs, enabling the admin to track login activity, changes made to the statistics.

6.2 Programming Code

Page.jsx

```
"use client";

import { useState } from "react";
import { loginWithEmailAndPassword, loginWithGoogle, loginWithGithub } from "@/helpers/loginHelp";
import { useRouter } from "next/navigation";
import { Input } from "@/components/ui/input";
import { Button } from "@/components/ui/button";
import { Card, CardContent, CardHeader, CardTitle } from "@/components/ui/card";
import {
  Dialog,
  DialogTrigger,
  DialogContent,
  DialogTitle,
  DialogDescription,
} from "@/components/ui/dialog";
import Link from "next/link";
import { toast, ToastContainer } from "react-toastify";
import { auth, db } from "@/config/firebase";
import { sendPasswordResetEmail } from "firebase/auth";
import "react-toastify/dist/ReactToastify.css";
import Header from "@/components/sections/Header";

const toastOptions = {
  position: "top-right",
  autoClose: 3000,
  hideProgressBar: false,
  closeOnClick: true,
  pauseOnHover: true,
  draggable: true,
  theme: "dark",
};

const Login = () => {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState(null);
  const [isDialogOpen, setIsDialogOpen] = useState(false);
```

```
    if (!email) return;
    setIsLoading(true);

    try {
      await sendPasswordResetEmail(auth, email);

      toast.success("Password reset link sent to your email!"); // Show success toast
      setIsDialogOpen(false);
    } catch (error) {

      toast.error("Error sending password reset email "); // Show error toast
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="flex min-h-screen text-white">
      <ToastContainer theme="dark" />
      <Header />

      {/* Left side with welcome message (transparent to show DarkVeil background) */}
       <div className="hidden md:flex md:w-1/2 bg-transparent p-8 flex-col justify-center items-center">
        <div className="max-w-md mx-auto">
          <h1 className="text-5xl font-bold mb-6 text-white">Welcome Back !</h1>
          <p className="text-lg text-gray-300 mb-8">Glad you're back! Log in to continue your journey with us.</p>
          <div className="animate-pulse">
            <button className="border border-white/30 bg-black/20 hover:bg-black/30 text-white rounded-md px-6 py-2 transition-all duration-300">
              Skip the lag ?
            </button>
          </div>
        </div>
      </div>

      {/* Right side with login form */}
      <div className="w-full md:w-1/2 flex justify-center items-center p-8 relative overflow-hidden">
        <div className="absolute top-[-30%] right-[-10%] w-64 h-64 rounded-full bg-purple-700/30 blur-3xl"></div>
        <div className="absolute bottom-[-20%] left-[-10%] w-64 h-64 rounded-full bg-purple-900/20 blur-3xl"></div>

        <Card className="w-full max-w-md bg-[#1a1a2e]/80 border border-gray-700 shadow-2xl rounded-lg backdrop-blur-sm z-10">
```

```
        <CardHeader>
          <CardTitle className="text-center text-2xl font-bold text-white">Login</CardTitle>
          <p className="text-center text-gray-400 mt-2">Glad you're back!</p>
        </CardHeader>
        <CardContent className="space-y-4">
          {error && <p className="text-red-400 text-sm text-center">{error}</p>}
        <form onSubmit={handleLogin} className="space-y-5">
          <div className="space-y-3">
            <Input
              type="email"
              placeholder="Username"
              value={email}
              onChange={(e) => setEmail(e.target.value)}
                 className="bg-[#1a1a2e]/50 border-gray-700 text-white rounded-md px-4 py-3
focus:border-purple-500 focus:ring-purple-500"
              required
            />
            <Input
              type="password"
              placeholder="Password"
              value={password}
              onChange={(e) => setPassword(e.target.value)}
                 className="bg-[#1a1a2e]/50 border-gray-700 text-white rounded-md px-4 py-3
focus:border-purple-500 focus:ring-purple-500"
              required
            />
          </div>

          <div className="flex items-center">
            <input type="checkbox" id="remember" className="mr-2 h-4 w-4 rounded border-
gray-700 bg-[#1a1a2e]/50 text-purple-600 focus:ring-purple-500" />
                  <label htmlFor="remember" className="text-sm text-gray-400">Remember
me</label>
          </div>

          <Button
            type="submit"
              className="w-full bg-gradient-to-r from-blue-500 to-purple-600 hover:from-blue-
600 hover:to-purple-700 text-white font-semibold py-3 rounded-md transition-all duration-
300" >
            Login
          </Button>
        </form>

        <div className="relative my-6">
          <div className="absolute inset-0 flex items-center">
            <div className="w-full border-t border-gray-700"></div>
```

```
            </DialogDescription>
            <Input
              type="email"
              placeholder="Your Email"
              value={email}
              onChange={(e) => setEmail(e.target.value)}
               className="mb-4 bg-[#1a1a2e]/50 border-gray-700 text-white rounded-md px-4
py-3 focus:border-purple-500 focus:ring-purple-500"
            />
            <div className="flex justify-end gap-3 mt-6">
              <Button
                variant="secondary"
                onClick={() => setIsDialogOpen(false)}
                className="bg-gray-800 hover:bg-gray-700 text-white text-sm font-medium py-
2 px-4 rounded-md border border-gray-700"
              >
                Cancel
              </Button>
              <Button
                onClick={handlePasswordReset}
                disabled={isLoading}
               className={`${isLoading ? "bg-purple-500/50" : "bg-gradient-to-r from-blue-500
to-purple-600 hover:from-blue-600 hover:to-purple-700"} text-sm font-medium py-2 px-4
rounded-md text-white transition-all duration-300`}
              >
                {isLoading ? "Sending..." : "Send Link"}
              </Button>
            </div>
          </DialogContent>
        </Dialog>
      </div>
    </CardContent>
  </Card>
  </div>
 </div>
 );
};
export default Login;
```

Chat.jsx

```
"use client";
import { useState, useEffect, useRef, useCallback } from "react";
import { auth, firestore } from "@/config/firebase";
import {
 collection,
 query,
 orderBy,
```

```
  limit,
  addDoc,
  serverTimestamp,
  onSnapshot,
  deleteDoc,
  doc,
  getDocs,
  where,
  updateDoc
} from "firebase/firestore";
import { Button } from "@/components/ui/button";
import { Input } from "@/components/ui/input";
import { Prism as SyntaxHighlighter } from 'react-syntax-highlighter';
import { vscDarkPlus } from 'react-syntax-highlighter/dist/cjs/styles/prism';
import { ClipboardDocumentIcon, CheckIcon, PaperAirplaneIcon } from
'@heroicons/react/24/outline';
import { MessageSquarePlus, Send, Sparkles, Trash, Trash2, X, XCircle, Clock, FileText, Zap,
StopCircle, Brain, Bot, User, Copy, Check } from "lucide-react";

// Import the custom hooks
import { useChatStreaming } from '@/hooks/useChatStreaming';
import { useConversationMemory } from '@/hooks/useConversationMemory';
import { useMessageParser } from '@/hooks/useMessageParser';
import { useChatAnalytics } from '@/hooks/useChatAnalytics';

function Chatroom({ workspaceId, setIsChatOpen }) {
  const [messages, setMessages] = useState([]);
  const [newMessage, setNewMessage] = useState("");
  const [loading, setLoading] = useState(true);
  const [streamingMessage, setStreamingMessage] = useState(null);
  const [showSummary, setShowSummary] = useState(false);
  const [isConnected, setIsConnected] = useState(true);
   const [providerInfo, setProviderInfo] = useState({ provider: 'gemini', model: 'gemini-2.0-
flash' });

  const userId = auth.currentUser?.uid;
  const name = auth.currentUser?.displayName || "Anonymous";
  const conversationId = `workspace_${workspaceId}`;

  const messagesRef = collection(firestore, "messages");
  const messagesQuery = query(messagesRef, orderBy("createdAt"));
  const messagesEndRef = useRef(null);
  const inputRef = useRef(null);

  // Use the custom hooks
     const { streamChat, isStreaming, streamingError, cancelStreaming } =
useChatStreaming('/api/getChatResponse');
```

```
if (newMessage.trim() === "") return;
    const messageText = newMessage.trim();
    const imageUrl = auth.currentUser?.photoURL;
    const aiMatch = messageText.match(/@X(.+)/);
    const aiPrompt = aiMatch?.[1]?.trim();

    // Clear input immediately for better UX
    setNewMessage("");

    try {
      // Always send the user message first
      await addDoc(messagesRef, {
        text: messageText,
        createdAt: serverTimestamp(),
        imageUrl,
        userId,
        name,
        workspaceId,
      });

      // If there's an AI prompt, generate response
      if (aiPrompt) {
        await generateAIResponseWithStreaming(aiPrompt);
      }
    } catch (error) {
      console.error("Error sending message:", error);
      // Restore message if failed
      setNewMessage(messageText);
    }
  };

  const clearChat = async () => {
    if (!window.confirm("Are you sure you want to clear all messages? This action cannot be
undone.")) {
      return;
    }

    try {
      const querySnapshot = await getDocs(
        query(messagesRef, where("workspaceId", "==", workspaceId))
      );

      const deletePromises = querySnapshot.docs.map((docItem) =>
        deleteDoc(doc(messagesRef, docItem.id))
      );
      await Promise.all(deletePromises);
{isCurrentUser && (
```

```
          <div className="flex-shrink-0">
            <div className="w-8 h-8 rounded-full bg-gradient-to-br from-blue-400 to-purple-
500 flex items-center justify-center shadow-lg">
              <User className="h-4 w-4 text-white" />
            </div>
          </div>
        )}
      </div>
    </div>
  );
};

 if (loading) {
  return (
   <div className="flex items-center justify-center h-full text-gray-400">
    <div className="flex flex-col items-center gap-4">
     <div className="relative">
          <div  className="animate-spin  rounded-full  h-12  w-12  border-2  border-gray-
600"></div>
       <div className="animate-spin rounded-full h-12 w-12 border-t-2 border-indigo-500
absolute inset-0"></div>
     </div>
     <div className="text-center">
      <p className="text-lg font-medium text-gray-300">Loading chat...</p>
      <p className="text-sm text-gray-500">Fetching conversation history</p>
     </div>
    </div>
   </div>
  );
 }

 return (
  <div className="flex flex-col h-full bg-gradient-to-br from-gray-900 via-gray-800 to-gray-
900 backdrop-blur-sm border border-gray-600/50 rounded-2xl shadow-2xl overflow-hidden">

   {/* Enhanced Header */}
   <div className="flex justify-between items-center p-4 bg-gradient-to-r from-gray-900/95
to-gray-800/95 backdrop-blur-xl border-b border-gray-600/50 shadow-lg">
    <div className="flex items-center gap-4">
     <div className="flex items-center gap-3">
      <div className="w-10 h-10 rounded-xl bg-gradient-to-br from-indigo-500 to-purple-
600 flex items-center justify-center shadow-lg">
        <Brain className="h-5 w-5 text-white" />
      </div>
      <div>
      size="sm"
      className="text-red-400 hover:text-red-300 hover:bg-red-900/20"
```

```
        >
          <Trash className="h-4 w-4" />
          <span className="hidden sm:inline ml-1">Clear</span>
        </Button>

        <Button
          onClick={() => setIsChatOpen(false)}
          variant="ghost"
          size="sm"
          className="text-gray-400 hover:text-gray-300 hover:bg-gray-700/50"
        >
          <X className="h-4 w-4" />
        </Button>
      </div>
    </div>

    {/* Messages Container */}
    <div className="flex-1 overflow-y-auto p-4 space-y-6 bg-gradient-to-b from-gray-800/20 to-gray-900/40">
      {messages.length === 0 && !streamingMessage ? (
       <div className="flex flex-col items-center justify-center h-full text-center py-12">
         <div className="mb-6 relative">
            <div className="w-16 h-16 rounded-2xl bg-gradient-to-br from-indigo-500 to-purple-600 flex items-center justify-center shadow-xl">
             <MessageSquarePlus className="h-8 w-8 text-white" />
           </div>
            <div className="absolute -top-2 -right-2 w-6 h-6 bg-green-400 rounded-full flex items-center justify-center">
             <Sparkles className="h-3 w-3 text-white" />
           </div>
         </div>
                <h3 className="text-2xl font-bold text-gray-200 mb-2">Start Your AI Conversation</h3>
         <p className="text-gray-400 mb-6 max-w-md">
          Type <code className="bg-gray-800 px-2 py-1 rounded text-indigo-400">@</code> followed by your question to chat with our AI assistant
         </p>
         <div className="grid grid-cols-1 sm:grid-cols-3 gap-3 text-xs text-gray-500">
           <div className="flex items-center gap-2">
             <Zap className="h-4 w-4 text-indigo-400" />
             Real-time streaming
           </div>
           <div className="flex items-center gap-2">
             <Brain className="h-4 w-4 text-purple-400" />
             Conversation memory
             Enhanced AI Chat
           </span>
```

```
            {streamingError && (
              <span className="text-red-400">• {streamingError.message}</span>
            )}
          </div>
          <span className="hidden sm:inline">Enter to send • Shift+Enter for new line</span>
        </div>
      </form>
    </div>
  </div>
 );
}

export default Chatroom;
```

Dashboard.jsx

```
"use client";

import { useState, useEffect } from "react";
import { Globe, Lock, Loader2, Search } from "lucide-react";
import { useRouter } from "next/navigation";
import { auth, db } from "@/config/firebase";
import { collection, addDoc, getDocs, doc, setDoc, deleteDoc } from "firebase/firestore";
import { PlusCircle, Trash2 } from "lucide-react";
import { Button } from "@/components/ui/button";
import { Card, CardContent } from "@/components/ui/card";
import DashboardNavbar from "@/components/DashboardNavbar";
import Link from "next/link";
import { toast, ToastContainer } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";
import { motion } from "framer-motion";
import { Dialog, DialogTrigger, DialogContent, DialogTitle, DialogDescription } from
"@/components/ui/dialog";
import { Input } from "@/components/ui/input";
import ShowMembers from "@/components/Members";

const toastOptions = {
  position: "top-right",
  autoClose: 3000,
  hideProgressBar: false,
  closeOnClick: true,
  pauseOnHover: true,
  draggable: true,
  theme: "dark",
};

const Dashboard = () => {
  const [workspaces, setWorkspaces] = useState([]);
```

```
const [loading, setLoading] = useState(true);
const [isOpen, setIsOpen] = useState(false);
const [workspaceName, setWorkspaceName] = useState("");
const [isPublic, setIsPublic] = useState(true);
const [isCreating, setIsCreating] = useState(false);
const [deletingWorkspaceId, setDeletingWorkspaceId] = useState(null);
const [searchQuery, setSearchQuery] = useState("");
const router = useRouter();
const user = auth.currentUser;

useEffect(() => {
 if (!user) {
  router.push("/login");
  return;
 }

 const fetchWorkspaces = async () => {
  try {
   const querySnapshot = await getDocs(collection(db, "workspaces"));

   const workspaceData = await Promise.all(
    querySnapshot.docs.map(async (workspaceDoc) => {
     const membersRef = collection(db, `workspaces/${workspaceDoc.id}/members`);
     const membersSnapshot = await getDocs(membersRef);

     const userMemberData = membersSnapshot.docs.find(
      (doc) => doc.data().userId === user.uid
     );

     if (!userMemberData) return null;

     return {
      id: workspaceDoc.id,
      ...workspaceDoc.data(),
      role: userMemberData.data().role || "Unknown",
     };
    })
   );

   setWorkspaces(workspaceData.filter(Boolean));
   setLoading(false);
  } catch (error) {
   console.error("Error fetching workspaces:", error);
   setLoading(false);
  }
   <span>Are you sure you want to delete this workspace?</span>
   <div className="flex space-x-2">
```

```
        <Button
          onClick={async () => {
            try {
              setDeletingWorkspaceId(workspaceId);
              await deleteDoc(doc(db, `workspaces/${workspaceId}`));
              setWorkspaces(workspaces.filter((ws) => ws.id !== workspaceId));
              toast.success("Workspace deleted successfully!", toastOptions);
            } catch (error) {
              toast.error("Failed to delete workspace.", toastOptions);
            } finally {
              setDeletingWorkspaceId(null);
              toast.dismiss(confirmationToast);
            }
          }}
          className="bg-red-600 hover:bg-red-500 text-white"
          disabled={deletingWorkspaceId === workspaceId}
        >
          {deletingWorkspaceId === workspaceId ? (
            <Loader2 className="h-4 w-4 animate-spin" />
          ) : (
            "Delete"
          )}
        </Button>
        <Button
          onClick={() => toast.dismiss(confirmationToast)}
          className="bg-gray-500 hover:bg-gray-600 text-white"
          disabled={deletingWorkspaceId === workspaceId}
        >
          Cancel
        </Button>
      </div>
    </div>,
    {
      ...toastOptions,
      autoClose: false,
      closeOnClick: false,
      draggable: false,
      hideProgressBar: true,
    }
  );
};

  const filteredWorkspaces = workspaces.filter((ws) =>
    ws.name.toLowerCase().includes(searchQuery.toLowerCase())
  );

        </motion.div>
```

```
            >
              {isCreating ? (
                <Loader2 className="h-5 w-5 animate-spin" />
              ) : (
                'Create Workspace'
              )}
            </Button>
          </div>
        </div>
      </DialogDescription>
    </DialogContent>
  </Dialog>
 </div>
 );
};
export default Dashboard;
```

Output.jsx

```
"use client";
import { forwardRef, useImperativeHandle, useState } from "react";
import { executeCode } from "../api";

const Output = forwardRef(({ editorRef, language }, ref) => {
 const [output, setOutput] = useState(null);
 const [isLoading, setIsLoading] = useState(false);
 const [isError, setIsError] = useState(false);

 const clear = () => {
  setOutput(null);
  setIsError(false);
 };

 const runCode = async () => {
  const sourceCode = editorRef.current?.getValue?.();
  if (!sourceCode) return;
  setIsLoading(true);
  try {
   const result = await executeCode(language, sourceCode);

   const out = [
    result?.run?.stdout && `Output:\n${result.run.stdout}`,
    result?.run?.stderr && `Runtime Error:\n${result.run.stderr}`,
   ]
    .filter(Boolean)
    .join("\n");

   setOutput(out ? out.split("\n") : ["No output"]);
   setIsError(!!result?.run?.stderr);
  } catch (error) {
   console.error(error);
        </div>
```

```
          <p className="text-gray-400">Run your program to see output here</p>
        </div>
      )}
    </div>
  </div>
 );
});

export default Output;
```

6.9 Implementation Details Summary

The implementation of XenAI – AI Powered Code Reviewer Using LLM was carried out through a modular and systematic approach integrating both frontend and backend technologies. The frontend was developed using Next.js to deliver a responsive, interactive, and developer-friendly workspace, while the backend was powered by Node.js and Firebase to handle authentication, database management, and API integrations. The AI core of XenAI utilizes the Google Generative AI (Gemini API) for intelligent code review, bug detection, and feedback generation, supported by LangChain for prompt orchestration and structured responses. The Firestore and Realtime Database were implemented to ensure persistent data storage and real-time collaboration among users. Git repository integration was achieved through the GitHub REST API, enabling users to push corrected code directly from the XenAI interface. The system underwent multiple phases of unit, integration, and user acceptance testing to validate performance and reliability.

CHAPTER 7

TESTING

The Software Testing phase of XenAI – AI Powered Code Reviewer Using LLM is a crucial stage that ensures the reliability, functionality, and performance of the system before deployment. Testing was carried out systematically to validate that each module such as authentication, AI code review, chat, and repository integration operates according to the defined requirements and delivers accurate results under different conditions.

The primary objective of this phase was to detect and rectify defects, verify functionality, and ensure that the AI-powered components provide correct and context-aware feedback to users.

7.1 Validation and System Testing

The validation process in XenAI was carried out after successful completion of unit and integration testing. Each module—such as Authentication, AI Review, Chat, Repo Integration, and Admin Dashboard—was verified against its functional and non-functional requirements. The Authentication Module was validated for secure login and access control using Firebase Authentication, ensuring that only authorized users could interact with the system. The AI Review Module was tested extensively using different programming languages and input sizes to validate that the Gemini API consistently generated accurate, context-aware code feedback.

Validation testing confirmed that the system fulfilled its intended purpose—providing developers with an intelligent, collaborative code review platform powered by AI. Additionally, performance testing was conducted to verify that the system maintained low latency and efficient response times during high user activity.

7.1.1 Software Testing

The Software Testing phase of XenAI – AI Powered Code Reviewer Using LLM is essential to ensure the system's accuracy, stability, and reliability across all modules. It involves systematically evaluating each component and its interactions to confirm that the software performs as expected under different scenarios.

Testing for XenAI was conducted in multiple levels — unit testing, integration testing, system testing, and acceptance testing. Each level focused on distinct aspects of quality assurance.

- **Unit Testing:** Every module of XenAI — including Authentication, Code Editor, AI Review Engine, and Chat Module — was tested individually to verify its functionality in isolation.
- **Integration Testing:** Once individual modules were verified, they were integrated to ensure seamless communication between the frontend, backend, and AI models. This

stage focused on testing data exchange between the Next.js interface, Firebase database, and Gemini API to ensure synchronization and consistency across the system.

- **System Testing:** The complete XenAI application was tested as a whole to evaluate the performance of all modules working together..
- **Acceptance Testing:** This final phase ensured that XenAI met its intended purpose and provided a satisfactory user experience. Testers and end-users evaluated the application's ease of use, responsiveness, and accuracy of AI feedback.

7.1.3 Reasons for Performing Software Validation

Software validation in *XenAI – AI Powered Code Reviewer Using LLM* plays a critical role in ensuring that the system developed truly fulfills its intended purpose and delivers consistent, reliable results to end users. The main goal of validation is to confirm that the software not only meets technical specifications but also aligns with user expectations, functional needs, and real-world performance standards.

One of the key reasons for performing validation in XenAI is to **ensure functional accuracy**. Since the system integrates multiple complex modules—such as AI-based code review using the Gemini API, real-time chat interaction, and repository integration—it is essential to validate that each function behaves exactly as intended. Validation ensures that when a developer submits code, the AI model analyzes it correctly, provides accurate feedback, and allows users to take appropriate corrective actions.

Validation is also performed to **ensure seamless integration** between various system components. XenAI's architecture connects a Next.js frontend, Firebase backend, and the Gemini AI engine. Validation testing confirms that data flows smoothly across these systems—ensuring that user requests trigger the correct AI responses, and that those results are accurately displayed in the interface without delays or loss of information.

Furthermore, **user experience and performance** are critical validation goals. The system is tested to confirm that it responds quickly to user actions, provides accurate AI suggestions, and maintains a stable session even under heavy load. This ensures that XenAI delivers an intuitive and reliable experience to developers using it for real-time code reviews. In conclusion, performing software validation in XenAI ensures that the application is **accurate, secure, efficient, and user-focused**, fulfilling both the technical and practical expectations of a next-generation AI-powered code review platform.

Test Cases

TC ID	Feature/Module	Test Scenario	Steps (brief)	Expected Result	Status
TC 1	**Authentication**	Login with valid cr edentials	Enter correct username and password	Successful login	
TC 2	**Code Editor**	Edit and Save	Make changes to code, click save	Code is saved with changes	
TC 3	**AI Review**	AI Suggestions	Submit code for review	Relevant suggestions	
TC 4	**Chat**	Multiple Models	Start chat, select various models	Chat works with all models	
TC 5	**Repo Integration**	Upload to Git	Save code, click upload button	Code successfully uploaded	
TC 6	**History**	View Past Reviews	Open history section	Previous reviews displayed	Pass
TC 7	**Error Handling**	Invalid Token	Attempt operation with bad token	Error message displayed	
TC 8	**Performance**	Concurrent Reviews	Shomit multiple code at once	All reviews processd correctly	Pass

Table 7.1:Test Cases

7.4 Testing Summary

The Testing Summary of XenAI – AI Powered Code Reviewer Using LLM provides a comprehensive overview of all the testing activities conducted to ensure that the system meets its intended functionality, performance, and quality standards. Testing was carried out systematically across all stages—from individual module verification to complete system validation—ensuring that the platform performs accurately, securely, and efficiently under different conditions.

Performance and security tests further validated XenAI's robustness. The system maintained high responsiveness even with multiple concurrent AI review requests, and user credentials were securely managed through Firebase Authentication and encrypted API tokens. Usability testing ensured that the interface remained intuitive and user-friendly, even during complex operations such as repository uploads or AI model switching.

CHAPTER 8

SAMPLE OUTPUT

8.1 Home page

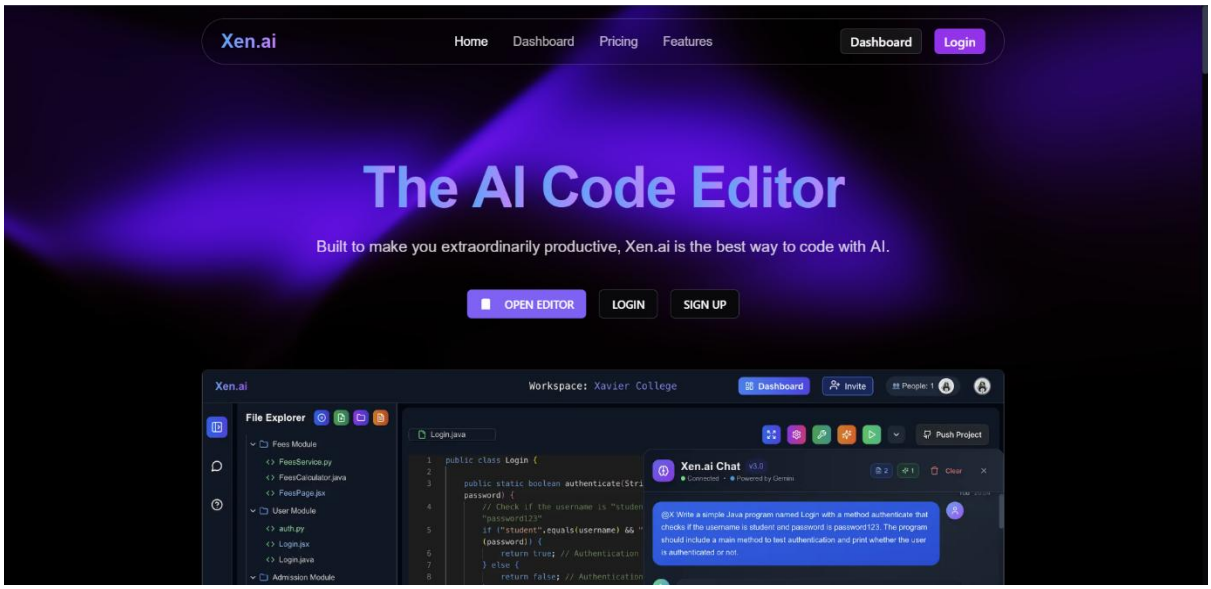
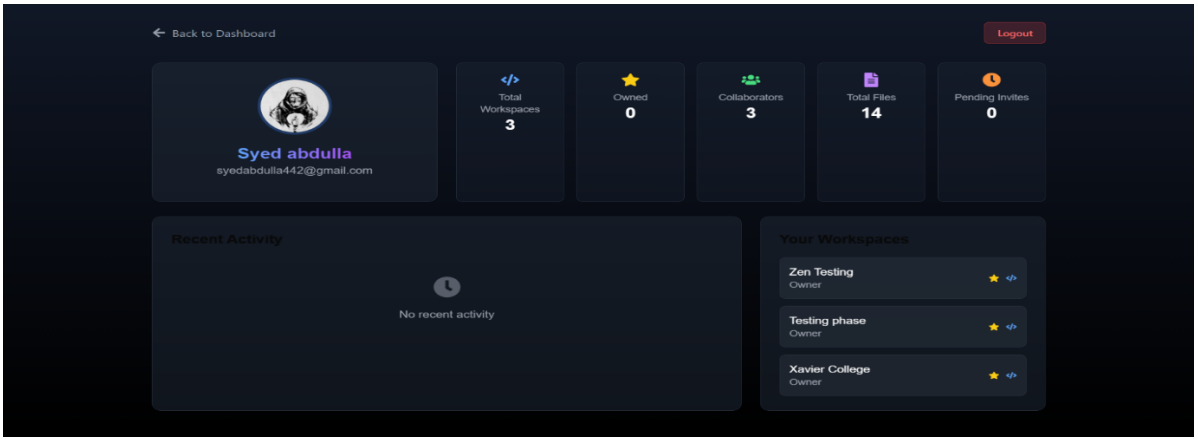


Fig: 8.1 Home page

8.2: User Dashboard and Workspace Overview



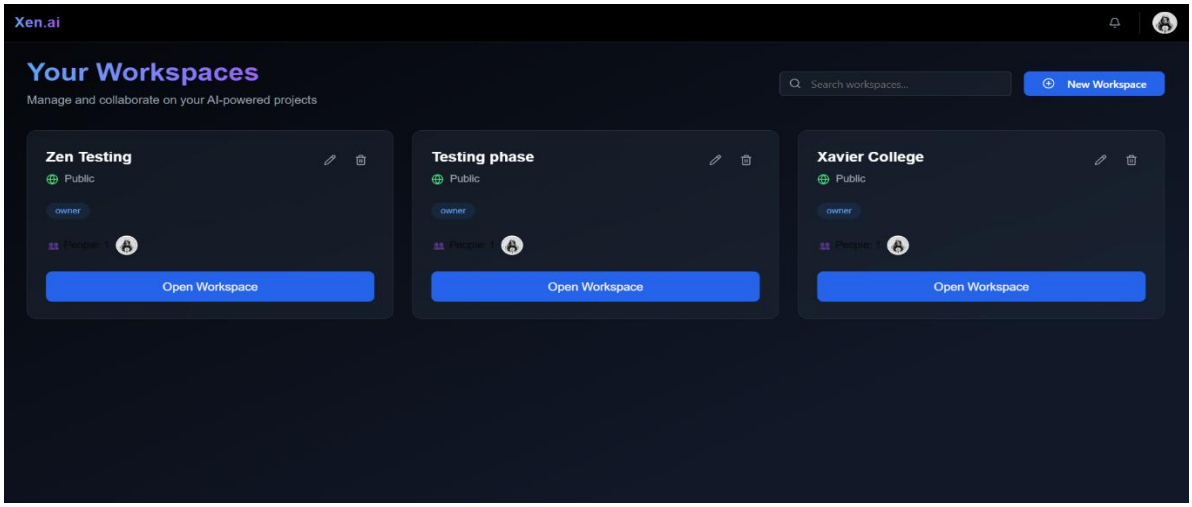


Fig : 8.2 User Dashboard and Workspace Overview

8.3: Code Editor Environment of XenAI

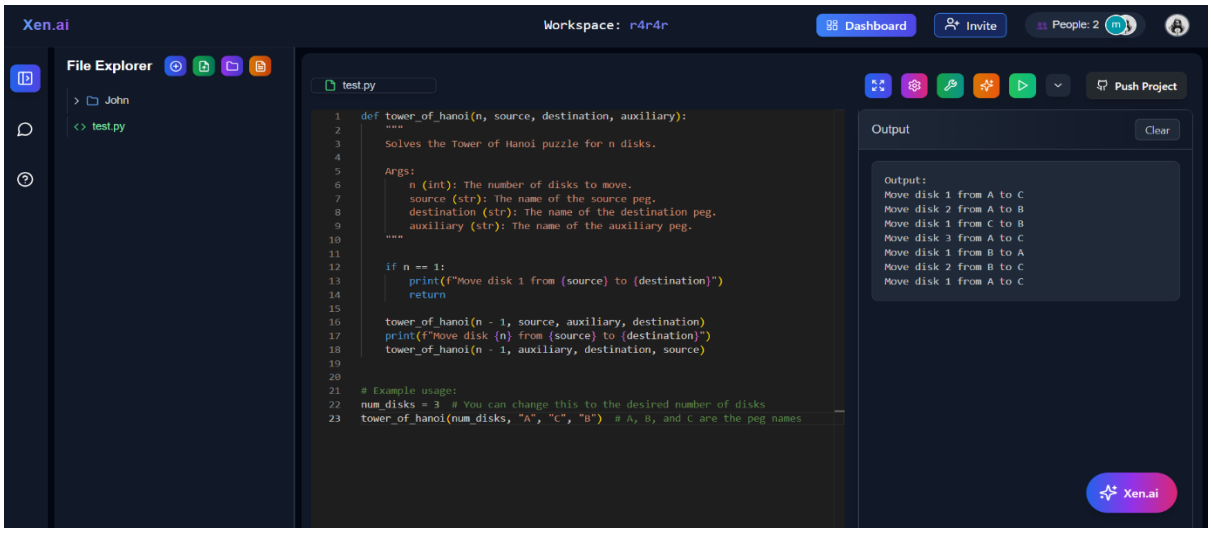


Fig:8.3: Code Editor Environment of XenAI

8.4: XenAI Chat Interface Powered by Gemini

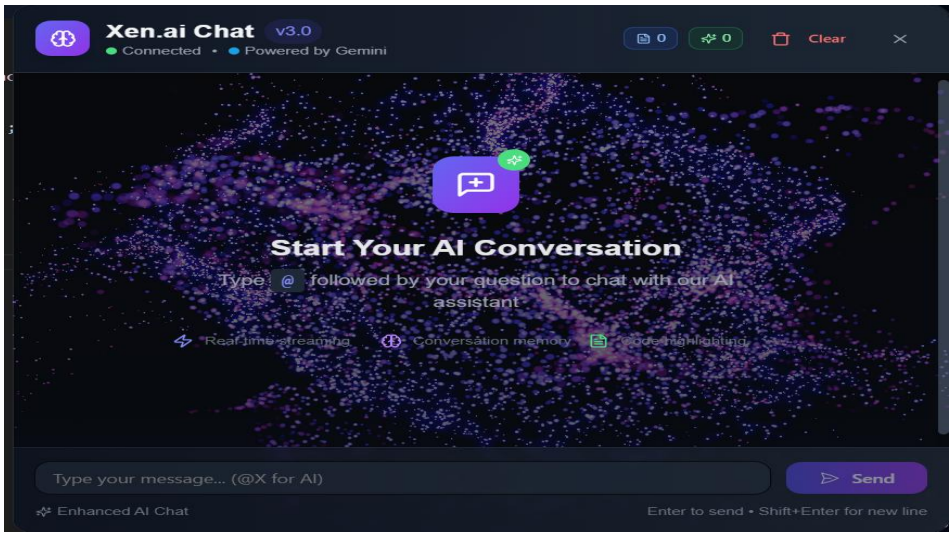


Fig: 8.4: XenAI Chat Interface Powered by Gemini

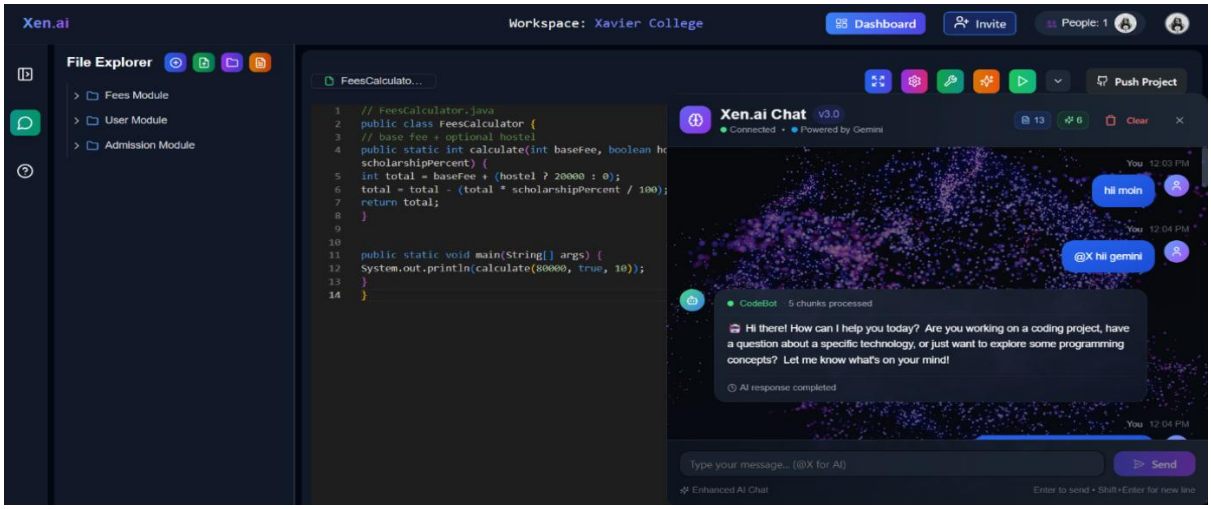


fig:8.5: Communicating Through XenAI Chatbot

8.5: Collaboration and Invite Feature in XenAI Workspace

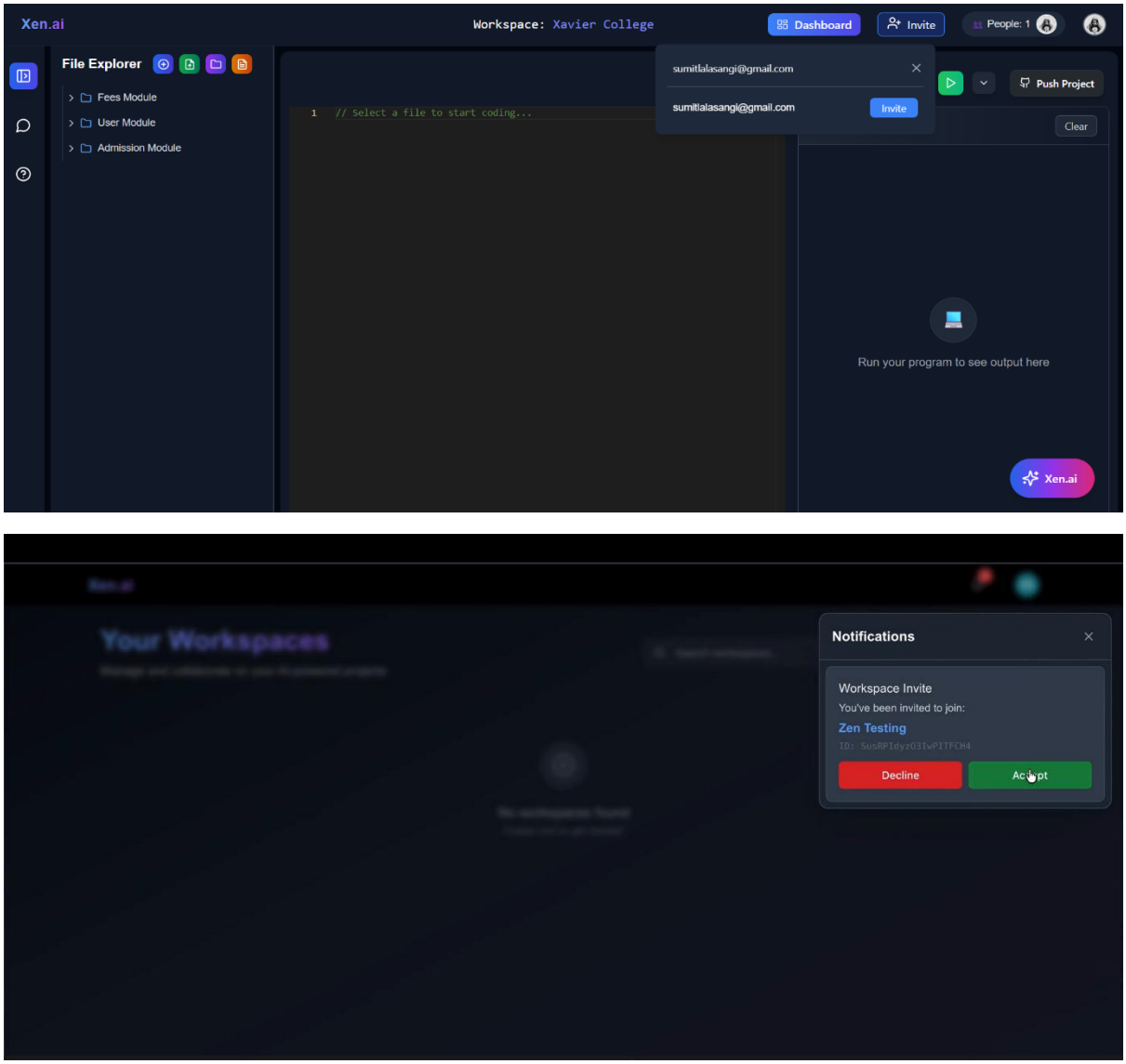


Figure 8.5: Collaboration and Invite Feature in XenAI Workspace

8.6: Git Repository Integration and Code Push Feature in XenAI

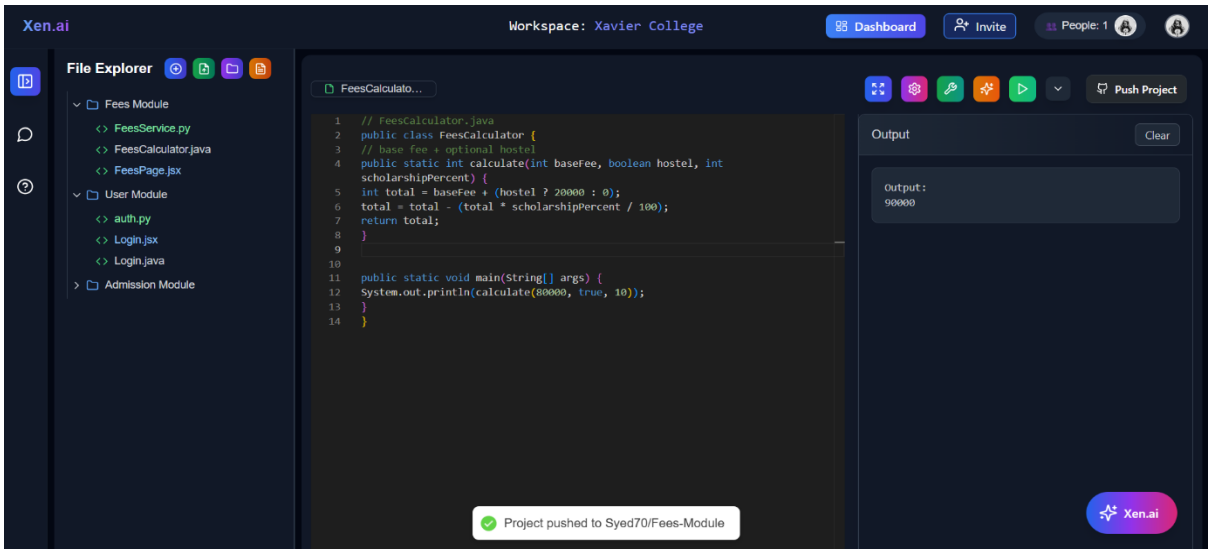


Figure 8.6: Git Repository Integration and Code Push Feature in XenAI

CONCLUSION

The development of XenAI – AI Powered Code Reviewer Using LLM marks a significant step toward integrating artificial intelligence into the modern software development lifecycle. The system successfully combines intelligent code analysis, real-time collaboration, and seamless integration with developer tools, making it a powerful and practical solution for improving code quality and productivity.

Throughout the project, each phase—from requirement gathering and design to implementation and testing—was carried out using the Agile methodology, ensuring continuous feedback and iterative improvement. The result is a platform that not only reviews and optimizes code but also assists developers through interactive AI chat, version control via Git integration, and secure authentication through Firebase.

By leveraging advanced LLMs (Large Language Models), XenAI provides precise code suggestions, identifies potential vulnerabilities, and helps developers understand and fix issues efficiently. Its modular architecture, comprising components like AI Review, Code Editor, Chat Module, and Repo Integration, ensures scalability and flexibility for future enhancements such as support for additional programming languages and CI/CD automation.

Future Scope

The future scope of XenAI – AI Powered Code Reviewer Using LLM is vast, with significant potential for expansion and innovation in the field of intelligent software development. In its future versions, XenAI can evolve into a fully autonomous development assistant capable of not only reviewing but also refactoring and deploying code across multiple environments. One of the primary areas of advancement lies in integrating multi-language support, enabling XenAI to review and optimize code in popular programming languages such as Java, C#, Python, and C++, thereby catering to a wider range of developers and organizations

REFERENCES

- S. Kumar, A. Sharma, and P. Gupta, "Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants," IEEE/ACM ICPC 2024. – Provided insights into evaluating AI-based code assistants, forming the base for XenAI's AI comparison model.
- R. Tiwari and V. Jain, "AICodeReview: Advancing Code Quality with AI-Enhanced Reviews," SoftwareX, 2024. This paper helped in understanding automated code review mechanisms and AI-based quality assurance, forming the basis for XenAI's intelligent review logic.
- P. Mehta, K. Sinha, and R. Verma, "AI-Based Code Review and Optimization System," IRJET, March 2025. This research guided the implementation of XenAI's optimization and suggestion engine using large language models to enhance code quality.
- A. Banerjee et al., "AI-Powered Code Review Assistant for Streamlining Pull Request Merging," ICWITE 2024. It served as a reference for integrating Git repository management and automating pull request handling in XenAI.
- L. Nair, M. Singh, and R. Kapoor, "CodeXchange: Leaping into the Future of AI-Powered Code Editing," ICCICA 2024. This work inspired the design of XenAI's Code Editor Module for real-time, collaborative, and intelligent code editing.
- S. Das and R. Patel, "AI-Powered Code Review and Vulnerability Detection in DevOps Pipelines," JSER, 2024. It supported the inclusion of secure code review mechanisms and vulnerability detection within XenAI's AI analysis framework.
- M. Johansson and E. Li, "Learning to Predict Code Review Completion Time in Modern Code Review," Empirical Software Engineering, 2023. This study contributed to improving workflow efficiency and review cycle prediction in XenAI.
- Compiled Survey, "Literature Survey 2," Internal Reference, 2024. This compilation provided a broad overview of existing AI-driven code review and NLP-based research, shaping the theoretical foundation of the project.
- Google Cloud, "Google Generative AI (Gemini API) Developer Guide," Google AI, 2024. This documentation was used to integrate the Gemini API for AI-based code understanding, review, and chat capabilities in XenAI.

Online References

• Google Cloud, “Generative AI with Gemini API – Developer Documentation,” Available at: https://cloud.google.com/vertex-ai/generative-ai
– Used for integrating Google’s Generative AI (Gemini) to power XenAI’s code analysis and chat modules.

• Firebase, “Firebase Documentation – Authentication and Firestore Database,” Available at: https://firebase.google.com/docs
– Referred for implementing secure authentication, real-time data management, and cloud connectivity.

• Next.js, “Next.js Framework Documentation,” Available at: https://nextjs.org/docs
– Used for building the frontend architecture and API routes of XenAI.

• LangChain, “LangChain Framework for LLM Integration,” Available at: https://www.langchain.com/
– Referred for managing communication and prompt handling between the LLM and backend.

• GitHub Docs, “REST API Reference,” Available at: https://docs.github.com/en/rest
– Used for implementing the repository upload and version control features in XenAI.