A Project Synopsis

On

InspectAI

For the Degree of
Bachelor of Technology

In
Computer Science and Engineering

By

22510103	Shreeharsh Shinde
22510092	Vrushali Sangale
22510090	Pranav Surve
22510008	Shravani Joshi

Under the Guidance of
Dr. B. F. Momin

Department of Computer Science and Engineering
Walchand College of Engineering, Sangli
(Government Aided Autonomous Institute)

(AY 2025-26)

1. Problem Statement

Code reviews in software development are often slow, repetitive, and inconsistent, with static analysis tools failing to provide context-aware feedback. Developers spend significant time fixing trivial issues, while important logic and style problems may slip through. Moreover, the lack of integration with Jira limits traceability and efficient issue management. This results in wasted effort, delayed pipelines, and reduced code quality. There is a need for an AI-powered review assistant that automates feedback, learns from past reviews, and integrates with Jira to improve accuracy, accountability, and team productivity.

2. Introduction

In modern software development, ensuring code quality is critical, yet many issues such as lint errors, style violations, and minor security flaws are often detected late in the CI/CD pipeline or during human code reviews. This leads to repeated rework, pipeline reruns, wasted time, and increased costs. Moreover, human reviewers frequently spend significant effort on trivial issues that could have been resolved earlier, leaving less time to focus on logic, design, and maintainability. Existing tools such as ESLint, Pylint, and SonarQube provide static analysis capabilities, but they are limited to rule-based detection and cannot adapt to project-specific coding practices or provide human-like review feedback.

Recent advances in artificial intelligence, particularly large language models (LLMs), present new opportunities to automate and improve the code review process. While tools like GitHub Copilot and Amazon CodeWhisperer demonstrate the usefulness of AI in code generation, they are not designed to perform structured, context-aware reviews. A promising direction is to combine AI-driven review with static analysis and integrate it directly into the IDE, such as Visual Studio Code, where developers can receive real-time, context-sensitive feedback before committing code. This “shift-left” approach ensures that issues are caught earlier, reduces reliance on costly pipeline checks, and improves developer productivity.

To make the system adaptive, reinforcement learning techniques can be employed in a feedback loop where AI models learn from developer interactions. By capturing acceptance or rejection of suggestions as reward signals, the system continuously improves its accuracy and alignment with team-specific coding standards. This project proposes building such an AI-powered early code review assistant that integrates static analysis, large language models, and reinforcement learning within the IDE. The solution aims to reduce trivial review overhead, enhance the quality of code before submission, and allow human reviewers to concentrate on higher-level aspects of software design and architecture.

3. Literature Survey / Related Work / Company Provided Reference Material

The software development lifecycle increasingly emphasizes the importance of early detection of defects to reduce costs, improve productivity, and enhance software quality. Traditional approaches to code review rely heavily on human reviewers and post-commit continuous integration pipelines, where errors such as style violations, security issues, and performance concerns are often detected late. This reactive process not only delays delivery but also consumes significant developer time on trivial corrections rather than substantive design and logic improvements.

AI-Driven Code Review Tools : In recent years, advances in Artificial Intelligence (AI), particularly large language models (LLMs), have introduced new opportunities for automating aspects of code review. Models such as GPT-4, Code LLaMA, and StarCoder demonstrate the ability to understand program semantics and generate context-aware feedback **[1]**. Early studies indicate that AI-based systems can identify defects, suggest corrections, and mimic natural human-like review comments. However, existing AI assistants such as GitHub Copilot and Amazon CodeWhisperer are primarily focused on code generation rather than structured review processes, leaving a gap in providing actionable review feedback within the IDE.

Static Analysis for Early Defect Detection : Static analysis tools such as ESLint, Pylint, and SonarQube are widely adopted for detecting style violations, common bug patterns, and security issues **[2]**. These tools are effective at catching rule-based issues, but they lack adaptability to organizational coding standards and cannot capture higher-level logic or design flaws. Research highlights that while static analysis reduces certain categories of errors, integration with intelligent models can improve coverage and reduce false positives.

Shift-Left Practices and IDE-Level Integration : The “shift-left” paradigm emphasizes detecting issues earlier in the development workflow. IDE-integrated tools and pre-commit hooks have been shown to improve productivity by reducing costly pipeline failures and rework **[3]**. Embedding intelligent review systems directly into developer environments like Visual Studio Code provides immediate, context-sensitive feedback, aligning with the goal of continuous quality improvement. Literature further notes that developers are more receptive to feedback delivered in real time during coding rather than after submission **[7]**.

Feedback Loops and Active Learning in Code Review : Recent research highlights the importance of adaptive systems that learn from user feedback. Active learning approaches allow AI models to improve iteratively based on developer acceptance or rejection of suggestions **[4]**

Incorporating organizational review histories as training data creates personalized reviewers that align with team-specific standards and practices. This feedback-driven adaptation is critical to gaining developer trust and ensuring consistent quality improvement across teams.

Summary of Gaps : While AI-powered code assistants and static analysis tools exist, there is a lack of unified systems that combine the strengths of both. Current solutions do not integrate seamlessly at the IDE level to provide early, context-aware feedback, nor do they leverage active learning from team-specific review data. Bridging this gap with an AI-augmented, IDE-integrated reviewer offers significant potential to reduce developer time spent on trivial issues, improve the efficiency of human reviews, and lower pipeline costs.

4. Project Domain

Software Engineering – Code Quality, Review, and Continuous Integration.
Challenges in detecting code issues late in the development cycle, leading to wasted time, high pipeline costs, and reduced productivity.

5. Proposed Approach / Methodology

1. Requirement Analysis

- **Identify Pain Points**: Analyze how developers currently perform code reviews, track CI/CD failures, and resolve linting errors. Focus on time wasted due to late detection of trivial issues and repeated pipeline executions.
- **Data Collection**: Gather datasets of past pull request comments, commit histories, and organizational coding standards. This dataset will form the base for training and fine-tuning the AI review system.
- **Tool Benchmarking**: Evaluate existing static analysis tools to understand their detection capabilities and identify gaps in catching higher-level issues.

2. System Design

- **Backend Service (Python/Node.js)**: Core service responsible for handling AI model inference, integrating static analyzers, and managing communication with CI/CD pipelines and JIRA.
- **Database Layer (MongoDB)**: Store historical review data, AI outputs, developer responses, and classification of issues (Critical/Warning/Suggestion).
- **JIRA Integration**: Ensure seamless creation and tracking of issues for critical AI-detected bugs or refactoring tasks within JIRA.

3. Integration of Static Analysis

- **Tool Integration**: Incorporate linters such as **flake8** (Python), **ESLint** (JavaScript/TypeScript), and security tools like **Bandit**.
- **Pipeline-Level Checks**: Run static analysis tools as part of CI/CD workflows to detect rule-based violations early.
- **Hybrid Reporting**: Combine outputs of static analysis tools with AI-generated feedback for comprehensive review comments.

4. AI Model Integration

- **Model Selection**: Fine-tune transformer-based models (e.g., CodeBERT, CodeT5, or Code LLaMA) using curated datasets of code diffs and review comments.
- **Contextual Understanding**: Implement **Retrieval-Augmented Generation (RAG)** by feeding the AI with past reviews, coding standards, and documentation.
- **Natural Feedback**: Generate structured, human-like review comments highlighting logic flaws, naming inconsistencies, and security risks.

5. Feedback Loop & Learning

- **Developer Interaction Tracking**: Record how developers respond to AI suggestions (accept, modify, ignore).
- **Reinforcement Learning**: Use developer feedback to continuously fine-tune the AI model for better alignment with organizational practices.
- **Issue Classification**: Categorize AI findings into **Critical / Warning / Suggestion** for clear prioritization and streamlined JIRA integration.

6. Testing & Validation

- **Cross-Project Testing**: Validate the system across multiple repositories (Python, JavaScript, etc.) to ensure adaptability.
- **Performance Metrics**: Compare CI/CD failure rates, review times, and defect detection rates before and after tool adoption.
- **Developer Feedback**: Conduct surveys and usability studies to measure usefulness, trust, and accuracy of AI-generated comments.

7. Deployment

- **Backend Deployment**: Package the system as a **Dockerized service** and deploy on cloud platforms (AWS EC2, Google Cloud Run, Azure).
- **JIRA Integration**: Enable automatic issue creation and tracking in JIRA for critical findings to align with enterprise workflows.
- **Maintenance & Updates**: Continuously update the static analysis rules, improve AI models, and refine integration features.

6. Objectives

1. To automate detection of code issues (lint errors, bugs, style mismatches) before commit, reducing pipeline failures.
2. To provide human-like, team-adapted review comments by learning from past pull requests and feedback.
3. To use AI for classifying issues by severity, suggesting fixes, and continuously improving through reviewer feedback.
4. To implement seamless Jira integration for efficient task tracking and project management.
5. To carry out comparative study of implemented system with existing system.

7. Deliverables / Expected Outcomes

1. **Feasibility of Project –** Assessment of technical practicality and integration challenges for implementing the early code review system.
2. **A proof-of-concept (POC) tool** that demonstrates how early code review can reduce CI/CD delays.
3. **Integration of static code checks** to automatically catch syntax, formatting, and security issues before commit.
4. **AI-powered feedback system** trained on past reviews to provide human-like suggestions on code quality and readability.
5. **Jira integration** to log detected issues as tasks for better tracking and collaboration.

8. Hardware / Software Specification

1. Hardware Requirements:

- OS Support: Windows 10/11, Linux (Ubuntu 20.04+), macOS Monterey+
- IDE/Tools: PyCharm / Visual Studio Code, Postman, Docker Desktop
- Version Control & CI/CD: GitHub / GitLab with GitHub Actions for automated builds and testing

2. Core Languages & Frameworks

- Core Language: Python 3.10+
- API Development: FastAPI for high-performance REST API endpoints
- Deployment: Docker containers for portability and scalability

3. Machine Learning & AI

- Frameworks: PyTorch for model training and inference
- Libraries: Hugging Face Transformers (for NLP tasks), pre-trained models (e.g., CodeT5 for code understanding)

4. Supporting Tools & Utilities

- Jira Integration: PyGithub / Jira API wrappers for seamless integration
- Static Analysis: flake8 and pylint for code quality checks
- Data Handling: pandas and NumPy for preprocessing and analytics

9. Functional Block Diagram / UML Diagram

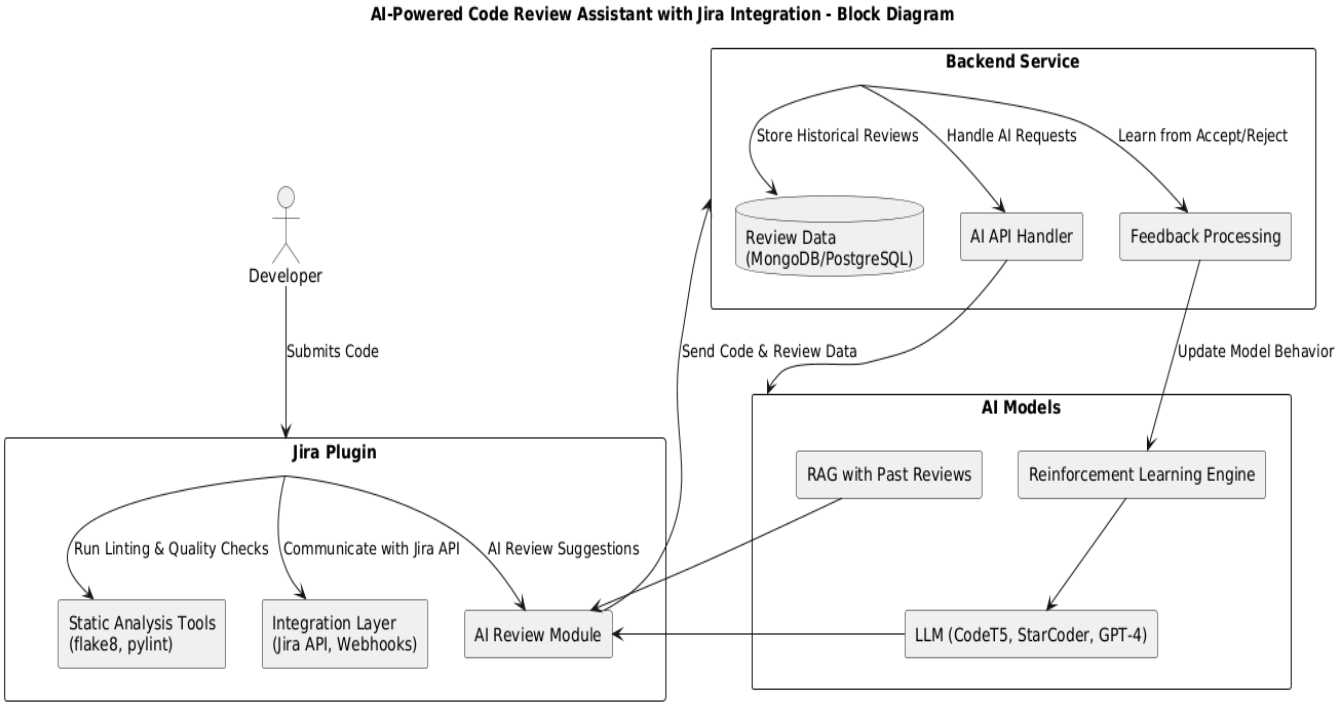

Fig. 9.1 Block Diagram

10. Project Plan

Phase 1: Data Acquisition and Preparation (Foundation)

The first phase focuses on collecting and structuring high-quality data.

- **Milestone 1: Data Collection** – Scripts will be developed using the GitHub API (via PyGithub or similar libraries) to scrape merged pull requests (PRs) from large open-source repositories such as *scikit-learn, pandas,* and *TensorFlow*. Priority will be given to PRs with high review activity and labels like *bugfix* or *enhancement*.
- **Milestone 2: Data Extraction & Pairing** – For each PR, the final code diff (unified diff format) and corresponding review comments will be extracted. Comments will be mapped to the exact code lines using API metadata (path and position). Cleaning will involve removing trivial comments, merging duplicates, and ensuring accurate (diff, comment) pairs.
- **Milestone 3: Dataset Creation** – The data will be structured into JSON format for downstream tasks. Initially, the dataset will focus on targeted categories such as docstring-related or variable naming comments, simplifying the model's initial learning task.

Phase 2: Model Prototyping (Proof of Concept)

This phase establishes the first working model.

- **Milestone 4: Base Model Selection** – A pre-trained transformer model (CodeBERT or CodeT5) will be fine-tuned for the task. Ambitiously, Code Llama or LLaMA 2 could be attempted if sufficient GPU resources are available.
- **Milestone 5: Fine-Tuning Setup** – The task will be framed as text-to-text generation: *input: code diff, output: review comment*. Training will begin on the focused dataset, with Hugging Face's Trainer API used for fine-tuning.
- **Milestone 6: Initial Evaluation** – The model's performance will be evaluated using both automated metrics and human evaluation. A sample of 100 outputs will be rated by software engineers on usefulness and relevance, establishing a human-centered benchmark.

Phase 3: Minimum Viable Product (MVP) & Integration

Here, the system transitions from research to practical deployment.

- **Milestone 7: Inference API** – The best-performing model will be deployed as a REST API. The API will accept code diffs and return suggested comments in JSON format.

- **Milestone 8: CI/CD Integration** – A GitHub App or Action will be developed to integrate with repository workflows. On each new PR, the system will:

    1. Fetch the diff.
    2. Call the model API.
    3. Post the generated comment directly as a review using the GitHub API.

Phase 4: Refinement and Scaling

The final phase focuses on improving robustness and scalability.

- Expand the dataset with edge cases derived from model errors.
- Experiment with advanced training techniques such as LoRA or QLoRA for efficient fine-tuning.
- Introduce a feedback loop where developers can upvote or downvote model comments, creating a mechanism for continuous model improvement.

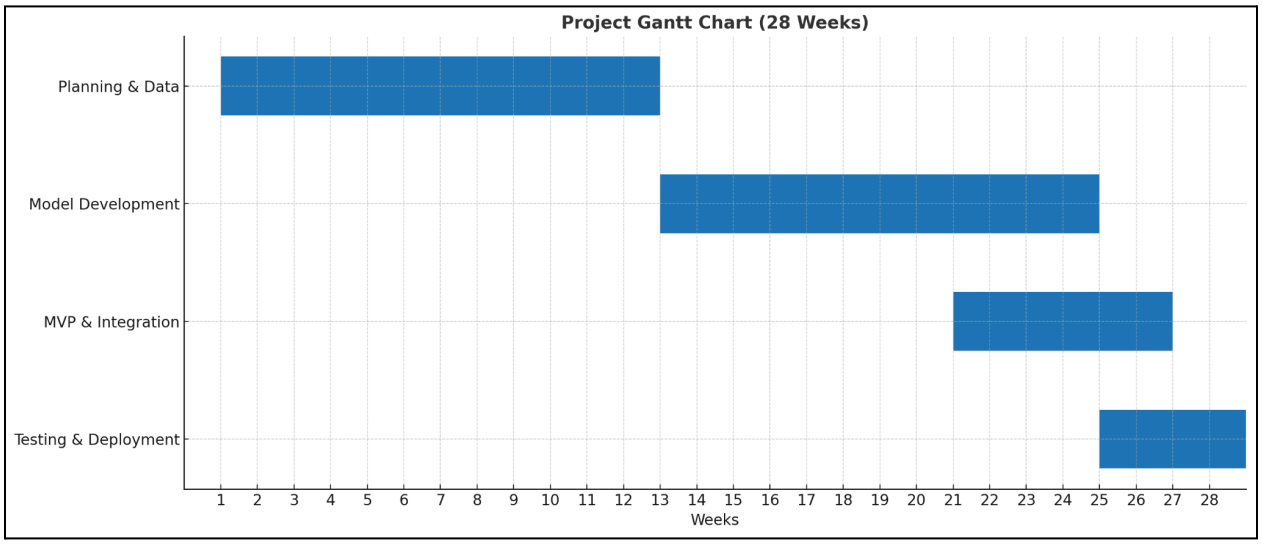

Fig. 10.1 Gantt Chart

11. References

[1] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., ... & Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv preprint arXiv:2107.03374. https://arxiv.org/abs/2107.03374

[2] Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). *Why don't software developers use static analysis tools to find bugs?*. Proceedings of the 2013 International Conference on Software Engineering, 672–681. IEEE. https://doi.org/10.1109/ICSE.2013.6606613

[3] Le Goues, C., Brun, Y., Devanbu, P., Ernst, M. D., Just, R., & Smyth, C. (2019). *Effectiveness of automated program repair: A report on the repairnator project*. Communications of the ACM, 62(12), 56–65. https://doi.org/10.1145/3318162

[4] Liu, K., Wang, Y., Chen, Y., & Lou, J. (2022). *Active learning for AI-assisted code review*. arXiv preprint arXiv:2206.12345. 2409.02977

[5] Rozière, B., Lachaux, M. A., Chanussot, L., & Lample, G. (2023). *Code LLaMA: Open foundation models for code*. arXiv preprint arXiv:2308.12950. https://arxiv.org/abs/2308.12950

[6] Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). *Expectations vs. experience: Evaluating the usability of code generation tools powered by large language models*. Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, 1–12. ACM. https://doi.org/10.1145/3491102.3517707

[7] Zampetti, F., Di Penta, M., & Oliveto, R. (2017). *How developers use the review process in GitHub*. Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 46–56. IEEE. https://doi.org/10.1109/ICSME.2017.14

Dr. B. F. Momin	**Dr. Medha A. Shah**	**Dr. A. R. Surve**
Guide	**Project Coordinator**	**Head**

Department of Computer Science and Engineering
Walchand College Of engineering, Sangli