IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155213

An AI-Powered Automated Code Review System Using Large Language Models and Static Analysis

Mrs. Nita Meshram[1], Ganni Naveen Raj Anudeep[2], K Vedavyas[3], K Harsha Nandhan[4], C Balaji Naidu[5]

Associate Professor, Department of Computer Science and Engineering,

K. S. School of Engineering and Management, Bengaluru, India[1]

UG Student, Department of Computer Science and Engineering,

K. S. School of Engineering and Management, Bengaluru, India[2-5]

Abstract: Code review is a critical phase in the software development lifecycle, ensuring code quality, maintainability, and security. However, manual code reviews are labor-intensive, often requiring 2 to 4 hours per pull request, and are highly susceptible to human error. Junior developers may inadvertently miss critical security vulnerabilities. This paper proposes an advanced AI-powered automated code review system that seamlessly integrates traditional rule-based static analysis with the deep contextual reasoning capabilities of Large Language Models (LLMs). Operating as an event-driven microservice, the system automatically triggers upon the creation of a GitHub Pull Request, passing Python code changes through a tri-layered analysis engine: Bandit for security vulnerability detection, Pylint for coding standard enforcement, and the Groq LLM for complex logical review and contextual feedback. Results are aggregated, mathematically ranked by severity, and posted directly to the pull request as a structured comment within a 60-second execution window. Deployed on a cost-effective stack including Render, Neon PostgreSQL, and FastAPI, this framework reduces review bottlenecks, minimizes security flaws, and enhances developer productivity.

Keywords: Automated Code Review, Large Language Models, Static Analysis, Continuous Integration, Software Security, Artificial Intelligence, GitHub Actions.

I. INTRODUCTION

The rapid acceleration of software delivery demands robust Continuous Integration and Continuous Deployment (CI/CD) pipelines. At the heart of these pipelines lies the code review process—a gatekeeping mechanism designed to catch bugs, enforce architectural standards, and prevent security vulnerabilities from reaching production environments. Historically, this process has been entirely manual. Developers submit a Pull Request (PR), and peer engineers meticulously read through the modified codebase to identify anomalies.

While manual peer review fosters knowledge sharing, it has become a severe bottleneck in agile development environments. Industry interactions and preliminary surveys reveal that manual code reviews consume an average of 2 to 4 hours per PR. Furthermore, the effectiveness of a manual review is directly proportional to the reviewer's expertise and current cognitive load. Under strict project deadlines, developers often experience "review fatigue," leading to superficial checks where only syntax or formatting issues are flagged, while deep logical or security flaws slip through. It is estimated that up to 70% of post-deployment security incidents in software teams can be traced back to oversights during the code review phase.

To mitigate human error, the industry widely adopted Static Application Security Testing (SAST) tools and linters. Tools like Pylint and Bandit are highly efficient at parsing Abstract Syntax Trees (AST) to find predefined anti-patterns. However, these tools operate in a vacuum; they lack the contextual awareness of the application's overarching business logic. Consequently, they often generate a high volume of false positives, leading developers to ignore their output entirely.

The advent of Large Language Models (LLMs) has opened new frontiers in software engineering. Models fine-tuned on programming languages exhibit a remarkable ability to understand code intent, suggest refactoring, and identify logical

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155213

bugs that traditional static analyzers miss. Yet, relying exclusively on LLMs for code review introduces its own challenges, including high computational costs, high latency, and the risk of "hallucinations"—where the AI confidently suggests incorrect fixes.

This paper proposes a novel, hybrid architecture that bridges the gap between deterministic static analysis and non-deterministic AI reasoning. By orchestrating Bandit, Pylint, and the Groq LLM (via LangChain) into a unified, event-driven pipeline, the system provides instantaneous, highly accurate, and context-aware feedback directly within the GitHub ecosystem.

The primary contributions of this paper are:

- The design and implementation of a zero-touch, automated PR review system that triggers via GitHub webhooks.
- A hybrid analysis methodology that combines the deterministic security and quality checks of Bandit and Pylint with the contextual reasoning of Groq LLM.
- A severity-ranking aggregation algorithm that prioritizes critical vulnerabilities and posts human-readable, actionable feedback within 60 seconds.
- A highly scalable, cost-effective deployment architecture utilizing open-source tools and cloud-native databases (Neon PostgreSQL) for long-term audit and quality reporting.

II. BACKGROUND AND RELATED WORK

A. The Evolution of Code Review

Code review transitioned from formal, synchronous meetings to asynchronous, tool-based reviews with the rise of distributed version control systems like Git. Platforms such as GitHub and GitLab popularized the Pull Request model, where code changes are isolated and discussed before merging. Despite the tooling improvements, the core task remained manual.

B. Static Analysis in CI/CD

Static analysis involves examining source code without executing it. Linters (e.g., Pylint, Flake8) enforce coding standards (PEP-8 in Python) and catch syntactical anomalies. Security scanners (e.g., Bandit, SonarQube) use pattern matching and AST traversal to detect known vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection, and hardcoded credentials. While fast and deterministic, static tools suffer from rigidity. They cannot infer whether a specifically flagged variable is safely sanitized in another un-scanned module, resulting in "noise" that degrades the developer experience.

C. Large Language Models for Code

LLMs, particularly those based on the Transformer architecture, have been extensively trained on massive repositories of open-source code. These models can perform complex software engineering tasks, including code generation, summarization, and translation. Recent studies have explored using LLMs for automated bug fixing and vulnerability detection. However, deploying them as standalone reviewers in CI/CD pipelines often results in prohibitive API costs and unacceptable latency.

III. LITERATURE SURVEY

To establish the context of our research, a comprehensive literature survey was conducted, analyzing the state-of-the-art in AI-driven software engineering. The four student authors collaboratively researched and compiled the data for this section. Table I presents a structured comparative analysis of recent frameworks attempting to automate code review, highlighting the specific gaps our proposed architecture aims to fill.

TABLE I
Comparative Analysis of Reviewed Works

Paper	Year	Methodology / Tools	Key Focus	Limitations	Proposed Solution
Kochhar et al. [1]	2025	Enterprise LLMs	Automating parts of code review workflows	Requires massive infrastructure; restricted to proprietary setups	Lightweight, cost-effective system built entirely on open APIs

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155213

Icoz & Biricik [2]	2025	LLMs + Symbolic Reasoning	Deeper logical code analysis	Computationally expensive; unsuitable for real-time PRs	Prioritizes rapid inference (under 60s) using the high-speed Groq API
Abtahi & Azim [3]	2025	Static Analysis + LLMs	Improving vulnerability detection and code understanding	Complex data processing pipelines to combine mixed outputs	Direct, rule-to-prompt aggregation without complex reconciliation
Chen et al. [4]	2025	Program Analysis + LLMs	General software understanding and code parsing tasks	Not specifically tailored as an automated PR review system	Applies methodology directly and exclusively to GitHub workflows
Sun et al. [5]	2025	BitsAI-CR (LLM Framework)	Comprehensive automated code review in industry practice	High cost and enterprise-scale deployment requirements	Fully deployable at zero cost using Render and Neon free tiers

IV. PROBLEM STATEMENT AND OBJECTIVES

A. Problem Formulation

Let P represent a Pull Request consisting of a set of modified files F = {f1, f2, ..., fn}. For each file fi, let Δfi denote the precise lines of code changed (the diff). The objective of the code review process R(Δfi) is to identify a set of defects D, where D can be categorized into security vulnerabilities (Dsec), quality issues (Dqual), and logical errors (Dlog).

Traditional manual review Rmanual is bounded by time constraints and human cognitive limits, leading to missed defects (Dmissed ⊂ D). Static analysis Rstatic successfully identifies a subset of Dsec and Dqual but introduces a high volume of false positives (Dfalse), while completely failing to identify Dlog.

Therefore, the problem is to design an automated function Rauto such that:

$$Rauto(\Delta fi) \approx Dsec \cup Dqual \cup Dlog$$

while simultaneously minimizing processing time T(Rauto) ≤ 60 seconds and minimizing the false positive rate.

B. System Objectives

To address the formulated problem, this project is guided by the following explicit objectives:

1) Automated Defect Detection: To automatically detect logical bugs, security flaws, and code quality degradation in GitHub pull requests without human intervention.
1) Unified Pipeline Integration: To seamlessly integrate static analysis tools (Bandit and Pylint) with generative AI (Groq LLM) into a single, cohesive analysis workflow.
2) Rapid and Contextual Feedback: To aggregate, deduplicate, severity-rank, and post the review comments directly onto the developer's pull request within a 60-second execution window.
3) Data Persistence and Auditing: To persist all review findings, metrics, and metadata in a PostgreSQL database for future auditing and project quality reporting.

V. PROPOSED METHODOLOGY AND SYSTEM ARCHITECTURE

The proposed system operates as an event-driven microservice. The architecture is completely decoupled from the developer's local environment, executing entirely in the cloud. The methodology is divided into five distinct operational phases.

A. Phase 1: Event-Driven Webhook Trigger

The system is anchored by a high-performance REST API built using FastAPI (Python 3.11). The repository on GitHub is configured to dispatch a POST webhook payload whenever a specific event occurs (e.g., pull_request.opened, pull_request.synchronize). Upon receiving the payload, the FastAPI endpoint verifies the cryptographic signature (using

DOI: 10.17148/IJARCCE.2026.155213

a shared HMAC secret) to ensure the request genuinely originated from GitHub. The payload contains critical metadata, including the repository name, the PR number, the base branch, and the commit SHA.

B. Phase 2: Code Ingestion and Diff Extraction

Once authenticated, the system utilizes the PyGithub API client to interact with the repository. Analyzing an entire repository for every PR is computationally wasteful and provides irrelevant feedback on legacy code. Therefore, the system targets only the modified code. The system extracts the Git Diff to identify the exact files altered. It filters this list to include only Python files (*.py). The raw content of these modified files, alongside the specific line numbers that were added or changed, is securely downloaded into a temporary, isolated runtime environment for processing.

C. Phase 3: The Hybrid Analysis Engine

This phase is the core computational engine of the framework. The ingested code is subjected to three parallel streams of analysis.

1) Security Analysis (Bandit): Bandit is executed against the temporary files. It builds an Abstract Syntax Tree (AST) of the Python code and runs a suite of security-focused plugins. It actively scans for SQL injection vectors (B601-B611), weak cryptographic configurations (B501-B507), hardcoded passwords (B101-B113), and Flask/Django misconfigurations (B201-B202). Bandit outputs a JSON report containing the severity (Low, Medium, High) and the exact line number of the violation.

2) Quality Analysis (Pylint): Simultaneously, Pylint evaluates the code against PEP-8 standards and general programming best practices. It flags architectural smells such as functions with too many branches (cyclomatic complexity), unused imports, missing docstrings, and variable shadowing. Pylint's output is parsed to extract the message ID, description, and line number.

3) Contextual AI Analysis (Groq LLM via LangChain): While Bandit and Pylint handle deterministic rules, the Groq LLM handles logical intent. The Groq API is utilized due to its novel LPU (Language Processing Unit) architecture, which provides inference speeds significantly faster than traditional GPU-based LLMs, ensuring the 60-second SLA is met. The LangChain framework dynamically constructs prompts injected with project context, the specific Git Diff, and the raw code of the modified functions. The LLM is instructed to output its findings in a strict JSON schema, identifying logical bugs, race conditions, edge-case mishandling, and suggesting optimized refactoring.

D. Phase 4: Aggregation and Severity Ranking

The outputs from the three parallel streams are vastly different in format and volume. The Aggregation Module normalizes these outputs into a unified data structure. To prevent overwhelming the developer, a mathematical severity score S is calculated for each identified issue to rank them. Let wtool be the weight of the tool (Security > Logic > Quality), and wconfidence be the confidence score returned by the tool (or LLM). The severity is calculated as:

$$S = (wtool \times \alpha) + (wconfidence \times \beta) + \gamma$$

Where α, β, and γ are tunable hyperparameters based on the team's specific requirements. Issues are then bucketed into Critical, High, Medium, and Low tiers. Duplicates (e.g., if both Pylint and the LLM flag an unused variable) are merged.

E. Phase 5: Feedback Generation and Persistence

The ranked issues are compiled into a highly readable, structured Markdown document. The system utilizes the GitHub Comments API to post this document directly to the PR thread. Critical issues are mapped to specific lines of code using GitHub's inline comment feature, allowing developers to see the exact context of the flaw. Concurrently, the raw analysis data, metadata (PR duration, author), and the generated findings are serialized using SQLAlchemy and committed to a Neon PostgreSQL database. This persistent storage allows engineering managers to query historical data and track code quality metrics over time.

VI. IMPLEMENTATION DETAILS AND TECHNOLOGY STACK

The system is designed for high availability and zero maintenance cost, leveraging modern cloud-native principles.

A. Software Requirements

The system runs on Python 3.11 and is cross-platform compatible (Windows 10/11, macOS 12+, Ubuntu 22.04 LTS). The backend framework is FastAPI for asynchronous request handling and high throughput. Analysis tools include Bandit (v1.7.5), Pylint (v3.0.3), and PyGithub (v2.1.1). AI orchestration is provided by LangChain (v0.1.0) and the Groq Python SDK. Database operations are managed via SQLAlchemy ORM with Alembic for automated schema migrations.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155213

B. Infrastructure and Deployment

The entire FastAPI application is containerized using Docker, ensuring that the runtime environment is identical across development and production. The Docker image is deployed on Render, a cloud application hosting platform that provides a free tier sufficient for handling standard webhook traffic from medium-sized repositories.

The database layer is hosted on Neon, a serverless PostgreSQL platform. Neon separates storage and compute, allowing the database to automatically scale down to zero during periods of inactivity, eliminating underlying infrastructure costs.

The CI/CD pipeline for the review tool itself is managed by GitHub Actions, triggering an automated build and deployment process to Render on any code update.

VII. EXPECTED RESULTS AND PERFORMANCE EVALUATION

A. Reduction in Review Latency

Traditional manual reviews require a developer to stop their current task, switch contexts, and read new code—a process taking hours. The proposed system operates asynchronously and instantly. Utilizing the Groq LLM (known for its sub-second LPU inference times), the entire pipeline—from webhook trigger to posted comment—is executed in under 60 seconds. This allows developers to receive immediate feedback on their PRs, enabling them to fix issues while the code is still fresh in their minds, drastically reducing the overall PR lifecycle.

B. Enhanced Vulnerability Detection

Manual reviews are notoriously poor at spotting complex security flaws like subtle SQL injection vectors or indirect cryptographic weaknesses. By utilizing Bandit, the system guarantees a deterministic catch rate for known Python CVEs (Common Vulnerabilities and Exposures). Furthermore, the LLM layer adds a heuristic check, capable of identifying business-logic flaws that standard security scanners miss (e.g., bypassing a custom authentication middleware). A reduction in undetected critical vulnerabilities of at least 80% is expected compared to purely manual reviews.

C. Noise Reduction and Developer Experience

A major complaint regarding traditional static analysis is the overwhelming number of trivial warnings (e.g., line too long, missing whitespace). By mathematically ranking the output and utilizing the LLM to filter out false positives based on context, the system ensures that the developer is presented only with high-value, actionable feedback. The inclusion of exact line numbers and LLM-generated code fix suggestions transforms the review from a simple "error report" into an interactive, educational tool.

VIII. LIMITATIONS AND FUTURE SCOPE

A. Current Limitations

Despite its advantages, the current framework has limitations. The system currently only supports Python codebases, as Bandit and Pylint are language-specific. Furthermore, while the Groq LLM provides rapid inference, it is still subject to token limits; extremely large pull requests containing thousands of lines of modified code may need to be truncated or chunked, which could cause the LLM to lose the broader architectural context.

B. Future Scope

Future iterations of this framework will focus on expanding language support. By integrating tools like SonarQube or Checkmarx, the system could easily adapt to support JavaScript, Java, and C++. Additionally, we plan to implement a feedback loop where the system learns from the developer's reactions. If a developer repeatedly dismisses a specific type of LLM suggestion, the system should dynamically adjust the LangChain prompt to suppress similar warnings in the future. Finally, integrating Retrieval-Augmented Generation (RAG) could allow the LLM to query the entire historical repository, ensuring that its suggestions perfectly align with the specific architectural patterns previously established by the development team.

IX. CONCLUSION

As software systems grow in complexity, the traditional manual code review process has proven to be unscalable, error-prone, and a significant bottleneck in CI/CD pipelines. Static analysis tools provide deterministic security checks but lack logical understanding, while standalone LLMs can be unpredictable and computationally expensive.

This paper successfully demonstrates the conceptualization and architecture of a unified, AI-powered automated code review system. By intelligently orchestrating Bandit, Pylint, and the high-speed Groq LLM within an event-driven FastAPI microservice, the proposed framework provides the best of both worlds. It delivers real-time, contextually

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155213

accurate, and severity-ranked feedback directly to GitHub Pull Requests in under 60 seconds. Deployed entirely on zero-cost, cloud-native infrastructure, this system not only hardens the security posture of Python applications by catching critical vulnerabilities early but also significantly alleviates developer fatigue, paving the way for more agile and resilient software engineering practices.

ACKNOWLEDGMENT

All four student authors (C Balaji Naidu, Ganni Naveen Raj Anudeep, K Vedavyas, and K Harsha Nandhan) equally contributed to the research, system architecture design, and the comprehensive literature survey presented in this paper. The project was conducted under the direct supervision and guidance of Mrs. Nita Meshram, Associate Professor, K. S. School of Engineering and Management, Bengaluru, India.

REFERENCES

[1]. P. S. Kochhar et al., "Automated Code Review Using Large Language Models at Ericsson," in Proc. IEEE Int. Conf. Software Engineering, 2025.
[2]. B. Icoz and G. Biricik, "Automated Code Review Using LLMs with Symbolic Reasoning," Journal of Systems and Software, 2025.
[3]. S. M. Abtahi and A. Azim, "Augmenting Large Language Models with Static Code Analysis," arXiv preprint arXiv:2506.10330, 2025.
[4]. X. Chen et al., "LLMs with Program Analysis for Code Understanding," ACM Transactions on Software Engineering and Methodology, 2025.
[5]. T. Sun et al., "BitsAI-CR: Automated Code Review via LLM in Practice," Proc. Automated Software Engineering Conference, 2025.
[6]. S. Tiangolo, "FastAPI Official Documentation," [Online]. Available: https://fastapi.tiangolo.com/
[7]. PyCQA, "Bandit Security Documentation," Python Code Quality Authority, [Online]. Available: https://bandit.readthedocs.io/
[8]. Groq Inc., "Groq API Documentation for High-Speed Inference," [Online]. Available: https://console.groq.com/docs

