AI-Based Reviewer for Project Code and Architecture: A Graph-Augmented Framework for Pull Request Review and Architectural Drift Detection

BaiXuan Zhang
Software Engineering Program
College of Arts, Media and Technology
Chiang Mai University
Chiang Mai, Thailand
Student ID: 652115546

Siraprapa Wattanakul
College of Arts, Media and Technology
Chiang Mai University
Chiang Mai, Thailand
Project Advisor

Abstract—Modern pull request (PR) review is increasingly constrained by repository scale, architectural complexity, and governance expectations. Prior studies show that review effectiveness depends on change understanding and reviewer context [1], while architecture-consistency research shows that implemented structures frequently drift from intended designs during software evolution [2], [3]. This paper presents AI-Based-Quality-Check-On-Project-Code-And-Architecture, a graph-augmented review framework that couples GitHub event ingestion, abstract syntax tree (AST) parsing, dependency-graph modeling, and configurable large language model (LLM) reasoning with governed project control. The framework (i) enriches pull request assessment with structural and repository context, (ii) detects circular dependencies, cross-layer violations, and coupling anomalies before they become deeply embedded in routine development, and (iii) records every review execution in an auditable, role-controlled workflow. A containerized prototype integrating a Next.js frontend, a FastAPI backend, PostgreSQL, Neo4j, and Redis provides a reproducible baseline for studying how AI-assisted review, graph-based architecture analysis, and software quality governance can be combined within a single engineering workflow.

Keywords—AI-assisted code review, pull request analysis, architectural drift detection, graph databases, software quality governance, large language models, dependency analysis

I. Introduction

Modern software teams depend on pull request workflows to preserve quality, share knowledge, and reduce regression risk. Prior work on modern code review shows that review effectiveness depends strongly on change understanding and reviewer context, both of which become harder to sustain as repositories and teams grow [1]. In parallel, research on architecture erosion and architecture consistency shows that implemented structures frequently drift away from intended design decisions during software evolution, creating maintainability and governance risk [2], [3].

This paper addresses the intersection of these two problems by presenting a graph-augmented review framework in which pull request analysis is coupled with dependency-aware architectural reasoning and governed project control. Review is therefore not limited to the textual diff: the workflow incorporates structural context from a dependency graph, repository events, and policy-oriented quality criteria. The paper makes three contributions: 1) a practical full-stack architecture that integrates AI-assisted pull request assessment with architectural analysis in one workflow; 2) an architecture-analysis pipeline that detects circular dependencies, cross-layer violations, and coupling anomalies through graph-based drift detection; and 3) an operational model that binds review execution to role-based access control, immutable audit logging, and reproducible project state. The remainder of this paper is organized as follows. Section III discusses related work, Section IV describes the proposed platform, Section V details its implementation, Section VI defines an evaluation framework that maps each contribution to observable evidence, and Sections VII and VIII discuss and conclude the work.

II. Research Context and Objectives

The research context is a web-based platform for programmers, reviewers, managers, and administrators who need faster review cycles without sacrificing traceability, security, or architectural visibility. In practice, these concerns remain fragmented across separate tools: code review platforms focus on line-level discussion, architecture tooling on structural inspection, and governance on administrative workflows. This fragmentation increases coordination effort and weakens the link between review decisions, architectural impact, and operational accountability. Two reference frameworks shape the design: ISO/IEC 25010:2023, which frames software quality through functional suitability, maintainability, reliability, and security [5], and the OWASP Top 10:2021, which keeps review and governance mechanisms attentive to common web-application risks [6].

Accordingly, the objective is not merely to automate comments on pull requests but to embed code review in a quality-governance loop: reduce review effort, surface architectural drift earlier in the lifecycle, and preserve operational evidence that supports later validation, auditing, and empirical study. Concretely, the work pursues three linked goals that correspond to the contributions above: review output that is structurally informed rather than purely text-based; architectural degradation that is visible during routine development rather than after late refactoring; and governed project operation enforced through access control, auditability, and reproducible platform state.

III. Related Work

Automated assistance for code review has progressed along complementary lines. Research on modern code review stresses that review effectiveness depends on reviewer context and change understanding [1]; in practice, most review still happens as diff-level discussion. Recent products such as GitHub Copilot code review [7] and Amazon CodeGuru Reviewer [8] automate part of this activity by generating

defect, best-practice, and security findings, while structure-aware code models such as GraphCodeBERT improve code understanding by learning from data flow rather than token sequences alone [9]. These tools, however, process the changed code largely in isolation: they do not relate a change to the dependency structure of the surrounding system, and their output is not framed by governance rules.

Architecture-consistency research shows that implemented structures drift from intended designs over time and that early detection reduces rework cost [2], [3]. Static analyzers such as SonarQube apply rule-based checks on each merge [10], and architecture validation is usually performed as a separate activity from day-to-day review. The framework proposed here is positioned at the intersection of these lines: it makes architecture analysis reactive to repository events, compares the current dependency graph against recorded baselines, and exposes drift indicators within the review itself, so that structural erosion and code-level findings are assessed together instead of in separate tool silos.

TABLE I. COMPARISON WITH EXISTING REVIEW TOOLING

Criterion	Copilot code review	CodeGuru Reviewer	Proposed framework
Structure-aware review context	No	No	Yes
Architectural drift detection	No	No	Yes
RBAC and immutable audit trail	No	No	Yes
Configurable LLM provider	No	No	Yes
Self-hosted deployment	No	No	Yes

IV. Proposed Platform

A. Architecture Overview

The platform follows a layered, event-driven architecture. The presentation layer uses Next.js to provide dashboards, review summaries, architecture views, and administrative workflows [13]. The application layer is implemented with FastAPI, which provides asynchronous request handling and structured API composition for authentication, repository integration, review orchestration, analytics, and settings management [12]. PostgreSQL stores transactional and governance data, Neo4j stores dependency graphs and baseline relationships, and Redis supports transient coordination for asynchronous analysis. GitHub webhook events trigger repository-driven workflows [11], and Cypher-based graph queries support structural reasoning over the architecture model [14]. Fig. 1 shows how interface, control, analysis, and data services cooperate within one governed pipeline: user-facing interaction, orchestration logic, review services, and persistent data stores are separated so that each layer can be inspected, replaced, and tested independently.

Fig. 1. Layered architecture of the proposed AI-Based quality-check framework, showing interface, control, analysis, and data-service responsibilities.

B. AI-Augmented Pull Request Review

When a connected repository emits a pull request event, the platform validates the webhook payload, acquires the relevant diff and metadata, parses changed files into abstract syntax trees, and packages local code evidence and project context for the review pipeline. A configurable LLM layer then produces ranked findings with severity labels, rationale, and remediation guidance; the use of LLMs is motivated by evidence that sufficiently scaled language models perform instruction-following and few-shot reasoning over textual inputs [4]. Model output, however, is grounded in repository metadata, structural context, and downstream governance rules rather than presented as unconstrained free-form advice. Fig. 2 depicts the complete event-driven path, from event receipt and context collection to storage and publication of structured results on the dashboard. To illustrate, consider a pull request that replaces a direct database call with a repository abstraction: the pipeline parses the changed files and the call graph, the LLM layer produces findings on the refactor and its intended behavior, and the graph analyzer simultaneously checks whether the modified edges still respect the recorded layering rules. The developer therefore receives, in a single governed response, both code-level and architecture-level feedback.

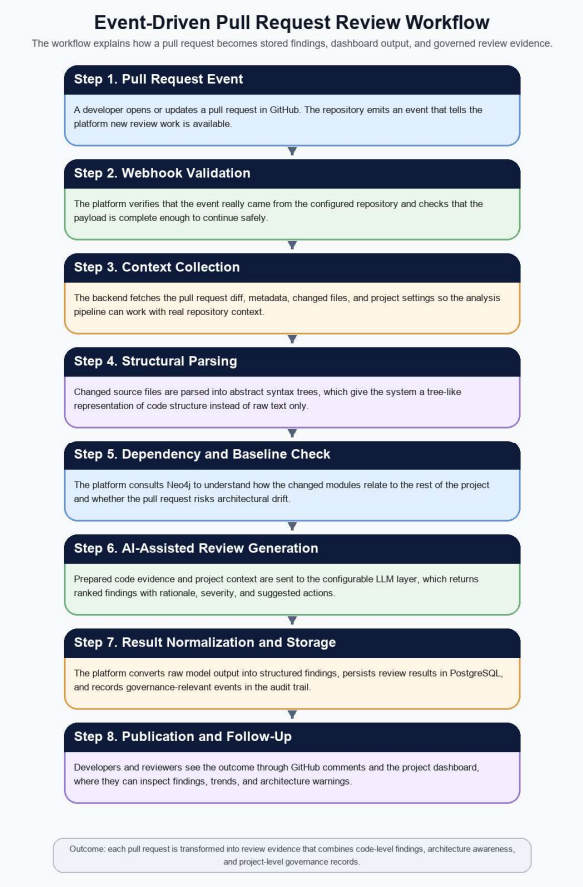

Fig. 2. Event-driven workflow for pull request analysis, structured finding generation, and governed publication of review output.

C. Graph-Based Architecture Analysis

The second core capability is architecture analysis driven by a project-specific dependency graph. Repository parsing extracts modules, classes, functions, and dependency edges, which are persisted in Neo4j for later traversal and comparison. This design choice follows the literature on architecture erosion and consistency, which emphasizes detecting drift as software evolves and maintaining alignment between intended and implemented structures [2], [3]. The platform compares the current graph against expected layering rules and previously captured baselines, and reports circular dependencies, unexpected cross-layer calls, coupling anomalies, and other drift indicators before they become deeply embedded in routine development.

D. Governance, Audit, and Project Control

Beyond analysis quality, the platform is designed for governed operation. Role-based access control separates visitor, programmer, reviewer, manager, and administrator responsibilities; project onboarding, provider configuration, role changes, and review execution are recorded in an immutable audit trail. This governance layer reflects the dual concern of quality and security: software quality objectives follow ISO/IEC 25010 [5], and security-sensitive review criteria follow OWASP Top 10 categories such as broken access control, cryptographic failures, and injection risks [6]. Review is therefore not comment generation but a controlled engineering activity: every triggered review can be traced to the actor, the settings active at execution time, and the way its result entered the project workflow.

V. Implementation Details

A. Full-Stack Application Layer

The current prototype is a containerized web application. The frontend is built with Next.js, React, and TypeScript to present review results, graph-based architecture views, and project-level metrics within one interaction model [13]. The backend is implemented with FastAPI and organized around endpoint groups for authentication, GitHub integration, pull request review, architecture analysis, analytics, settings, and audit access [12]. Keeping interaction, analysis logic, and persistent storage in separate layers preserves modular boundaries, reduces operational fragmentation, and makes the prototype easier to reproduce and extend.

B. Persistence and Analysis Services

PostgreSQL provides transactional consistency, auditable operational records, and project configuration state. Neo4j is reserved for structural analysis because graph traversal is central to dependency reasoning, drift detection, and baseline comparison [14]. Redis supports low-latency coordination for scheduling analysis jobs and invalidating cached state. The model-facing stage of the pipeline is provider-abstracted, so OpenAI-compatible or Anthropic-compatible services can be integrated without altering the orchestration flow: reasoning capability is model-dependent, whereas workflow control, governance logic, and evidence preservation remain platform responsibilities.

C. Deployment and Operational Baseline

The deployment baseline is intentionally pragmatic. Development and staging environments are containerized, health-checked, and isolated through service boundaries, so the prototype can be reproduced on local machines and hosted Linux environments with parity between demonstration and validation. The operational design also supports controlled experimentation: service boundaries localize failures, database state can be inspected independently of the review interface, and queue-backed workflows make it possible to study processing latency and orchestration behavior without collapsing the application into a purely synchronous request model.

VI. Evaluation Framework

The current work presents a research-oriented prototype rather than a completed benchmarking study. The evaluation framework therefore links each contribution to measurable evidence, so that future validation can demonstrate, rather than assert, the value of the design. Table I maps the four principal dimensions to the concrete platform mechanisms that implement them, the evidence sources that record them, and the outcomes that define success.

A. Research Questions

The evaluation is organized around three research questions, each aligned with one contribution and operationalized by the dimensions of Table I:

RQ1 (Review efficiency): does the framework reduce review turnaround time and produce consistently prioritized findings?
RQ2 (Drift detection): do graph-based baselines surface architectural drift earlier and more accurately than diff-only review?
RQ3 (Governance and traceability): does the framework preserve complete, auditable evidence of review execution?

The metrics and evidence sources listed in Table I are designed to answer these questions during future systematic validation.

B. Assessment Dimensions

Review efficiency is measured through turnaround time, comment quality, and prioritization consistency. Architectural governance is assessed through the framework's ability to identify drift patterns before they accumulate into large refactoring episodes. Operational traceability depends on the completeness of audit records, permission boundaries, and reproducibility of review actions. Usability and adoption depend on whether developers, reviewers, and managers can interpret findings, trends, and risk summaries without additional manual correlation, and whether outputs are trusted enough to influence routine practice. Together, these dimensions treat the prototype not only as a code-analysis tool but as a governed engineering workflow that can be evaluated technically (does it work?) and practically (can people act on its output?).

C. Target Outcomes

The intended outcomes are shorter review cycles, clearer prioritization of risky pull requests, improved visibility of architectural erosion, and stronger confidence in governed software delivery. In practical terms, the framework should reduce time spent on repetitive inspection, provide earlier correction signals to developers, and support managerial reasoning through quality dashboards and drift indicators rather than anecdotal reporting. A successful evaluation must therefore show not only that the system produces findings, but that those findings are timely, interpretable, and connected to decision making in real development workflows, particularly for AI-assisted tools, where raw automation without trust and usability rarely changes practice.

D. Threats and Limitations

Several limitations remain. LLM output quality may vary by provider, prompt design, and codebase characteristics. Very large monorepositories may stress graph storage and repeated parsing. Teams with strongly manual review cultures may require onboarding and calibration before trusting automated findings. Most importantly, this paper presents an implementation and an evaluation framework rather than a completed comparative experiment, so claims about measurable productivity gain remain future work rather than empirical conclusions. These limitations define the boundary conditions for future experimentation and longitudinal evaluation, and they position the framework as a structured research baseline rather than a finished universal solution.

TABLE II. EVALUATION DIMENSIONS OF THE PROPOSED PLATFORM

Dimension	Platform Mechanism	Evidence Source	Intended Outcome
Review efficiency	Webhook-triggered analysis, AST parsing, ranked AI findings	Pull request history, review timestamps, and issue logs	Reduced review turnaround time and more consistent triage
Architectural governance	Neo4j dependency graph, drift rules, and architecture baselines	Graph snapshots, drift reports, and architecture	Earlier detection of structural erosion and rule violations
		dashboards	
Operational traceability	RBAC, audit logging, settings management, and project scoping	Audit records, access logs, and settings change history	Explainable and reviewable governance evidence
Usability and adoption	Dashboard summaries, issue explanations, and project analytics	Scenario walkthroughs, stakeholder feedback, and demo sessions	Higher reviewer and manager confidence in automated support

Table I summarizes the proposed validation framework: it links each principal dimension to a concrete platform mechanism, the evidence source that can substantiate it, and the intended outcome, thereby turning the contributions introduced in Section I into checkable evaluation criteria.

VII. Discussion

The paper's central contribution is its integration strategy. Prior work emphasizes reviewer understanding in code review [1] and the cost of divergent architecture in consistency research [2], [3]; this framework connects both concerns by treating pull request review, dependency-aware architecture reasoning, and governance evidence as one operational workflow instead of separate tools or late-stage audits. Review quality is influenced not only by what changed in a pull request, but also by how the change affects structural design and whether the resulting decision can be traced and governed over time.

A methodological contribution complements the architectural one. Requirements, architecture, implementation, and verification artifacts are designed to be traceable to one another, improving reproducibility and strengthening the prototype as a subject for empirical study. This traceability is particularly valuable for AI-assisted engineering systems, where explanation quality, governance, and architectural impact matter as much as raw automation speed, and it provides a structured basis for studying AI-assisted review as a socio-technical workflow.

VIII. Conclusion

This paper presented AI-Based-Quality-Check-On-Project-Code-And-Architecture, a graph-augmented framework that combines GitHub-triggered automation, abstract syntax tree parsing, dependency-graph analysis, configurable LLM reasoning, and governed project access. The three contributions are supported by distinct parts of the design: the full-stack architecture (Sections IV and V) is the operational evidence of feasibility, the drift-detection pipeline (Sections IV-C and V-B) provides the structural analysis mechanism, and the governance model (Section IV-D) provides traceable, auditable review execution.

Within this paper, evidence is presented at the level of an implemented, containerized prototype and a defined evaluation framework (Section V) rather than as measured productivity results; the latter is deliberately scoped as future work. Subsequent work should therefore focus on controlled evaluation with real repositories, measurement of review-cycle improvements, calibration of model-generated findings, and refinement of explainability for high-severity architectural and security issues, as well as adoption behavior and the trade-off between automated guidance and human

oversight. In short, AI-assisted review becomes more valuable when connected to structural reasoning and governance evidence rather than treated as a standalone commenting tool.

REFERENCES

[1] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," in Proc. 35th Int. Conf. Softw. Eng. (ICSE), San Francisco, CA, USA, 2013, pp. 712-721, doi: 10.1109/ICSE.2013.6606617.

[2] L. R. De Silva and D. Balasubramaniam, "Controlling Software Architecture Erosion: A Survey," J. Syst. Softw., vol. 85, no. 1, pp. 132-151, Jan. 2012, doi: 10.1016/j.jss.2011.07.036.

[3] N. Ali, S. Baker, R. O'Crowley, S. Herold, and J. Buckley, "Architecture Consistency: State of the Practice, Challenges and Requirements," Empir. Softw. Eng., vol. 23, no. 1, pp. 224-258, 2018, doi: 10.1007/s10664-017-9515-3.

[4] T. Brown et al., "Language Models Are Few-Shot Learners," in Adv. Neural Inf. Process. Syst., vol. 33, 2020, pp. 1877-1901.

[5] ISO/IEC 25010:2023, Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — Product Quality Model, Nov. 2023.

[6] OWASP Foundation, "OWASP Top 10:2021," 2021. [Online]. Available: https://owasp.org/Top10/2021/. Accessed: Mar. 24, 2026.

[7] GitHub, Inc., "About GitHub Copilot code review," [Online]. Available: https://docs.github.com/copilot/code-review. Accessed: Mar. 24, 2026.

[8] Amazon Web Services, "Amazon CodeGuru Reviewer: User Guide," [Online]. Available: https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/. Accessed: Mar. 24, 2026.

[9] D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," in Proc. Int. Conf. Learn. Represent. (ICLR), 2021. [Online]. Available: https://arxiv.org/abs/2009.08366. Accessed: Mar. 24, 2026.

[10] SonarSource, "SonarQube Server Documentation," [Online]. Available: https://docs.sonarsource.com/sonarqube-server. Accessed: Mar. 24, 2026.

[11] GitHub Docs, "Webhook Events and Payloads," [Online]. Available: https://docs.github.com/en/webhooks/webhook-events-and-payloads. Accessed: Mar. 24, 2026.

[12] FastAPI, "FastAPI Documentation: First Steps," [Online]. Available: https://fastapi.tiangolo.com/tutorial/first-steps/. Accessed: Mar. 24, 2026.

[13] Vercel, "Next.js Documentation," [Online]. Available: https://nextjs.org/docs. Accessed: Mar. 24, 2026.

[14] Neo4j, Inc., "Cypher Manual," [Online]. Available: https://neo4j.com/docs/cypher-manual/current/. Accessed: Mar. 24, 2026.