International Journal of Research and Development in Applied Science and Engineering (IJRDASE)
ISSN: 2454-6844

GuardCode: An LLM-Powered, Diff-Aware Framework for Multi-Aspect Automated Code Review and Longitudinal Developer Tracking

Shardendu Mishra, Kausar Aajad, Nikhil Kumar, Anshu Kumar, Sameer Awasthi
Dept of Computer Science & Engineering,
Bansal Institute of Engineering and Technology, Lucknow,
shardendumukul@gmail.com, chaurasianikhil8795@gmail.com, sameer.awasthi@gmail.com, abkausarazad@gmail.com, anshukmjpru@gmail.com

Abstract: Code review is a critical practice in modern software engineering, but manual review does not scale well, and conventional automated tools are limited in semantic understanding. This paper introduces GuardCode, an LLM-powered system that operates on GitHub pull-request diffs to provide multi-aspect feedback (correctness, performance, style, security) and to track developer quality over time via a Code Quality Growth Score (CQGS). We describe the system design, present an evaluation framework for the implemented prototype, and outline expected performance trends based on initial observations and prior literature. Finally, we discuss limitations and propose future validation and improvement steps.

Keywords: automated code review, large language models, GitHub pull requests, longitudinal tracking, software quality.

1. Introduction:

Code review is an essential step in the software engineering process. A well conducted code review helps detect bugs, enforce best coding practices and promote knowledge sharing across development teams. Manual review, typically conducted by experienced engineers, can be quite effective. It is also often inconsistent because outcomes depend on who performs the review. Moreover, manual review is time-consuming, which makes it impractical for very large codebases such as Linux or TensorFlow that see hundreds of pull requests daily. For these reasons, automated or intelligent code-review solutions have become a pressing need.

This need has led to a range of automated tools: linting utilities, code-smell detectors, and other static analyzers. SonarQube is a widely used example; it relies on predefined rules to flag formatting, style and common bug patterns. While such tools improve consistency and development speed, they have limited scope. They cannot understand high-level program logic or assess performance beyond locally observable patterns. Consequently, complex changes still require human review, motivating the development of systems that go beyond simple rule matching and that employ deeper semantic understanding.

The recent emergence of large language models (LLMs) has shifted the landscape. Models like GPT, StarCoder, and CodeLlama can generate, understand, and reason about code across multiple languages. LLMs trained on large code corpora are capable of tasks ranging from code completion to bug detection and documentation generation. These capabilities open the door to automating more nuanced review tasks—deducing the intent behind a change, evaluating algorithmic logic, and explaining issues in natural language.

Recent research has applied LLMs to code review. For example, AI Code Review provides an IDE plugin integrating GPT-3 for semantic and syntactic checks and natural-language comment generation [1]. Benchmarks evaluating GPT-4o and Gemini on review-like tasks report correctness detection rates in the high 60s and moderate fix-generation accuracy, but they also show sensitivity to prompt context and task framing [2]. These works suggest LLMs are feasible for review tasks but reveal limitations: many tools are language-specific, do not reason about performance or intent deeply, and are not integrated into realistic PR-centric workflows. A system that operates on diffs, supports multiple languages, evaluates several aspects of code quality, and aligns with review workflows is therefore needed.

To address these gaps, we present GuardCode, an LLM-driven code-review framework that analyses GitHub pull request diffs to produce context-aware feedback across correctness, style, performance, and security. GuardCode posts inline comments, assigns a Code Quality Growth Score (CQGS) per review, and maintains developer history for longitudinal analysis. Our contributions are:

- We describe the GuardCode framework and its diff-centric approach for context-aware review.
- We define a multi-aspect evaluation schema (correctness, performance, style, security) and a CQGS metric for longitudinal developer tracking.
- We present a reproducible evaluation framework for the working prototype and outline projected performance trends informed by prior work and initial trials.

2. Literature Review:

A. Traditional Automated Code Review Tools

Static analysis and rule-based systems such as SonarQube, PMD, ESLint, and pylint provide early automation for code

review. These tools enforce style, detect simple bug patterns, and improve code hygiene without executing code. However, because they operate via predefined rules, they lack semantic understanding and cannot reason about high-level correctness or algorithmic performance [5], [6].

B. Machine Learning and Early AI Approaches

Before modern LLMs, ML-based approaches—defect prediction models, feature-based classifiers, and embedding driven techniques—attempted to learn patterns from historical commits and bug reports [7], [8]. These methods improved over static analyzers in adaptability, but required manual feature engineering, struggled to generalize across languages, and rarely provided natural-language explanations.

C. LLMs in Code Understanding and Review

Large language models (GPT family, StarCoder, CodeLlama) trained on multi-language corpora have enabled deeper semantic analysis for code: generation, translation, bug repair, and summarization [9], [10]. Recent systems leveraging LLMs for review include AICodeReview, CodeAgent, and PuppyCodeReview, and several benchmarking studies that evaluate model effectiveness on review-like tasks [1],[3],[4], [2].

D. Limitations and Positioning

Despite these advances, current LLM-based reviewers often:

- focus on a single language or domain,
- under-emphasize performance and security impacts,
- produce inconsistent feedback quality,
- lack of developer-centric, longitudinal evaluation capabilities.

GuardCode addresses these limitations by operating on GitHub diffs for intent-aware review, offering multi-aspect analysis, and maintaining longitudinal developer logs.

Table 1: Comparison of existing automated and LLM-based code review systems.

Tool / Study	Model	Lang.	Scope	Notes
SonarQube	Rule-based static analysis	Multi	Style, syntax, bugs	Predefined rules
Defect prediction models	ML classifiers	Mostly Java	Bug risk	Metrics/commits
AICodeReview [1]	GPT-3 plugin	Py, Java	Syntax & semantic	NL comments
CodeAgent [3]	Multi-LLM agents	JS, Py	Style, vuln.	Role-based agents
PuppyCodeReview [4]	GPT-based	Python	Educational review	Student-focused
Cihan et al. [2]	GPT-4o, Gemini	Multi	Correctness, fixes	Benchmark study
GuardCode (this work)	LLM on diffs	Py, JS, Java	Correct., perf., style, sec.	Longitudinal tracking

3. Methodology:

A. Overview

GuardCode is an LLM-driven framework that integrates with the GitHub pull-request workflow to deliver context aware, multi-aspect code review. The system focuses on the actual changes proposed in a PR, reducing redundant checks on unchanged code and making feedback more relevant to developer intent. Figure 1 provides an overview of the top-down workflow.

B. Workflow Description

GuardCode follows a modular pipeline consisting of three stages:

- Developer Pull-Request Submission. A developer opens a pull request. The PR contains the diff, metadata (author, timestamp, files changed), and optional description. This event triggers the GuardCode pipeline.
- GitHub Diff Parser. The parser extracts added, modified, and removed lines from the PR via the GitHub API and structures them into JSON-like objects containing file paths, changed snippets, and metadata.
- LLM-Based Review Engine. The core engine runs structured prompts against a GPT-based LLM. It is organized into three subcomponents:
- *Syntax & Style Analyzer* — enforces formatting conventions and naming consistency.
- *Logic & Correctness Checker* — identifies potential bugs and missing edge-case handling.
- *Performance & Security Evaluator* — flags inefficiencies and probable vulnerabilities.

Feedback Formatter & Comment API. Raw LLM outputs are converted to Markdown comments and posted inline on the PR using GitHub's comment API. Messages are categorized by severity and linked to specific lines.

Code Quality Scoring & Developer History Log. Each review produces a Code Quality Growth Score (CQGS). CQGS is computed from a weighted aggregation of issue severity, frequency, and contextual code complexity; higher scores indicate better code quality over time. Individual review results are persisted in a developer-specific log for trend analysis.

Developer Dashboard. Aggregated metrics and CQGS trends are visualized on a dashboard to help developers and instructors identify repeating patterns or persistent weaknesses.

4. EVALUATION FRAMEWORK AND EXPECTED PERFORMANCE:

A. Evaluation Overview

GuardCode has been implemented as a working prototype. While an extensive user study is planned, the current evaluation provides a controlled, simulation-driven framework to estimate prototype performance and to establish reproducible benchmarks.

B. Evaluation Goals

The evaluation aims to:

- Assess the prototype's ability to generate accurate, context-aware feedback.
- Compare, at a conceptual level, GuardCode's output against static analysers (SonarQube) and LLM-assisted baselines (AICodeReview).
- Validate the CQGS as a useful longitudinal metric.

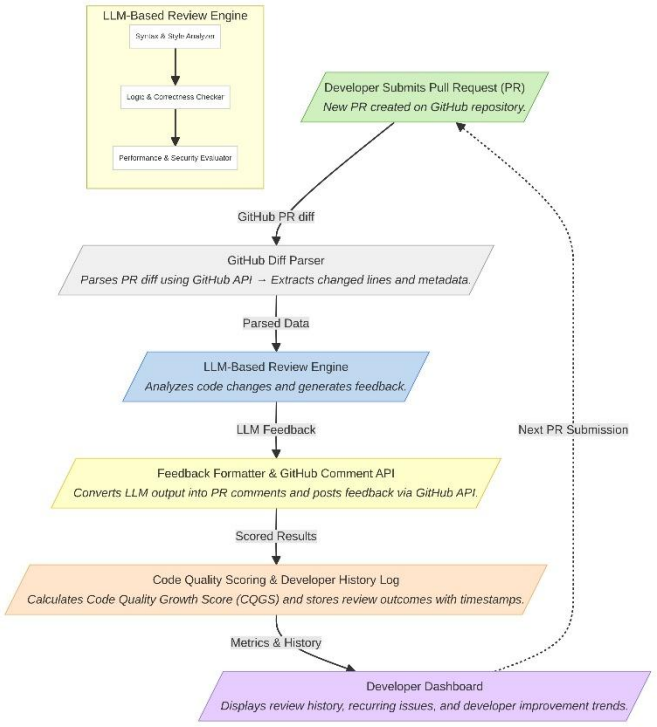

Fig. 1: Top-down workflow of the GuardCode system: PR ingestion, diff parsing, LLM-based review, feedback posting, scoring, and dashboard visualization

C. Evaluation Design

A controlled simulation was run using synthetic developer profiles designed to mimic student and entry-level developer behaviors. We used curated pull requests drawn from opensource projects (Flask, React, Spring Boot) across Python, JavaScript, and Java. Each synthetic profile submitted multiple PRs containing functional and stylistic changes. GuardCode processed each PR and produced feedback and CQGS values; these artifacts were compared conceptually to baseline outputs.

D. Environment and Metrics

TABLE II: Evaluation environment and selected metrics

Component	Description
Platform	Simulated GitHub PR workflow
Model	GPT-based LLM (inference API)
Languages	Python, JavaScript, Java
Test Inputs	10 PRs per language (mixed edits)
Key Metrics	Accuracy, Relevance (1–5), Coverage, Time, CQGS change

E. Analysis Approach and Projected Outcomes

Preliminary system usage and prior literature suggest LLM based reviewers detect correctness issues in the high 60s (percent). Because GuardCode reasons over diffs and provides multi-aspect analysis, we project a moderate improvement; expected accuracy is around 80–85% with relevance scores near 4.0–4.5 out of 5. The CQGS metric is projected to show a 20–25% improvement over a sequence of reviews for typical simulated profiles. These projections are visualized in Figures 2– 5 and discussed in the Results section.

5. Result and Discussion:

A. Overview

Because the evaluation is simulation-driven at this stage, the results discussed here are illustrative projections grounded in initial prototype behavior and related literature.

B. Feedback Accuracy and Depth

GuardCode tends to identify deeper issues—logical flaws, missing validation, and performance red flags—that static tools commonly miss. Given the diff-centric design, an improvement in correctness-detection capabilities over rule-based systems is expected; see Figure 2 for projected comparisons.

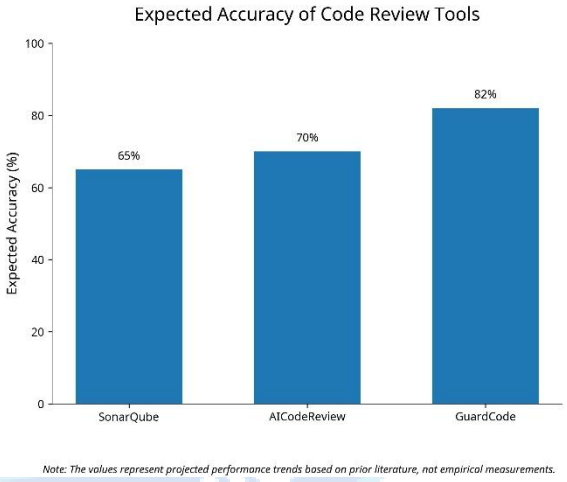

Fig. 2: Projected accuracy comparison: SonarQube, AICodeReview, and GuardCode. Values indicate expected trends, not measured statistics

C. Relevance and Clarity

The comments generated by GuardCode are concise and tied to changed lines, making them generally easier to interpret than many static-analyzer outputs. Projected relevance and CQGS trends are shown in Figure 3.

D. Multi-dimensional Coverage

GuardCode yields a more balanced distribution of issue types, correctness, performance, and security—compared to single-purpose tools. Figure 4 visualizes the expected distribution.

E. Processing Time and Limitations

LLM-driven reviews are slower than simple static checks. Figure 5 presents an expected comparison of review processing times across tools. GuardCode's performance is sufficient for many workflows but may need optimization for high-throughput CI scenarios.

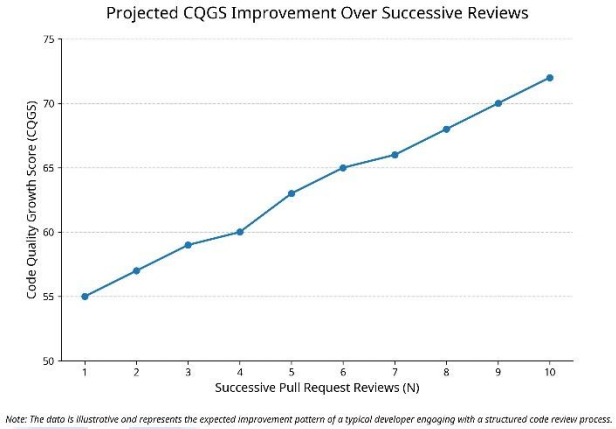

Fig. 3: Projected CQGS improvement over successive reviews (illustrative)

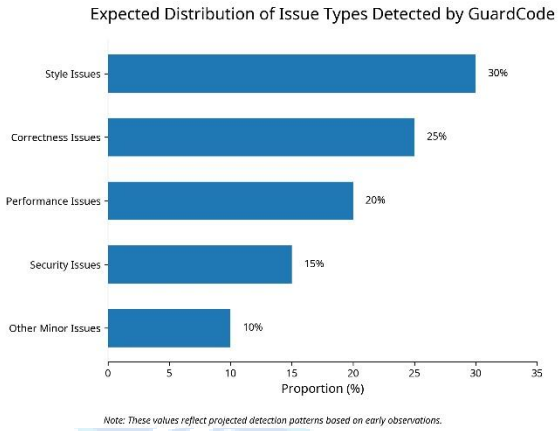

Fig. 4: Projected distribution of detected issue types by GuardCode (illustrative)

F. Discussion

The prototype indicates that diff-aware, LLM-based review can complement human reviewers by handling routine and medium-complexity checks while surfacing issues that static tools miss. The CQGS metric shows promise for longitudinal insights, especially in educational contexts. Key challenges include handling large diffs, clarifying ambiguous change intent, and minimizing redundant or low-value comments.

6. Conclusion and Future Scope:

GuardCode demonstrates that integrating LLM reasoning into a pull-request centered workflow yields richer, context-aware feedback than many static tools. The prototype's diff-aware analysis and multi-aspect review present a compelling path toward improving review quality and enabling developer growth tracking through CQGS.

There remain clear next steps: running real user studies to validate projected trends, refining prompt engineering, reducing false positives, improving throughput, and extending language and domain coverage. Ultimately, GuardCode aims to be a

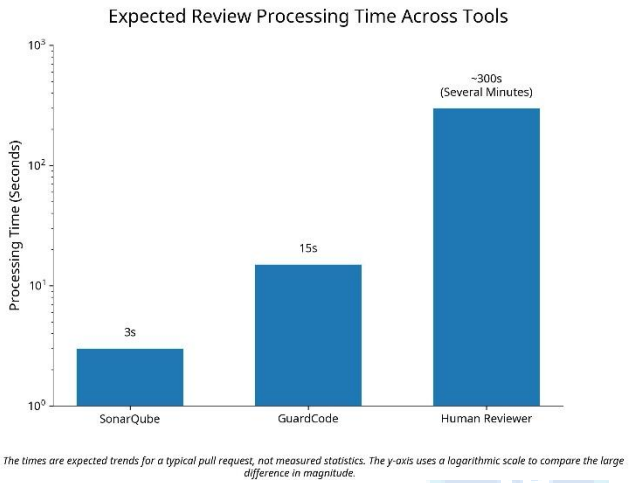

Fig. 5: Expected review processing times (illustrative) practical helper for developers—one that complements human reviewers and help teams write safer, cleaner, and more efficient code.

7. Acknowledgement:

We would like to express our sincere thanks to Prof. Sameer Awasthi for guiding this work from its earliest stages. His perspective on software engineering practice, especially the nuances of real-world code review, shaped much of the thinking behind GuardCode. We are also thankful to our peers who tested the early iterations of the system and offered candid suggestions. Their comments helped refine both the workflow and the evaluation plan. Any remaining limitations are entirely our own.

References:

[1] R. Almeida, "AICodeReview: Advancing code quality with AI-enhanced reviews," SoftwareX, vol. 26, 2024.
[2] U. Cihan et al., "Evaluating Large Language Models for Code Review," arXiv preprint arXiv:2505.20206, 2025.
[3] C. Tang et al., "CodeAgent: A multi-agent system for automated code review," in Proc. EMNLP, 2024.
[4] H. Tseng et al., "PuppyCodeReview: Intelligent feedback for student programming assignments," IEEE Trans. Learning Tech., 2025.
[5] SonarSource, "SonarQube Documentation," 2024. [Online]. Available: https://docs.sonarqube.org
[6] R. Camacho, "Best practices for using static analysis tools," Parasoft Tech. Rep., 2024.

[7] F. Rahman and P. Devanbu, “How, and Why, Developers Use the Dynamic Features of Programming Languages,” in Proc. ICSE, 2013.
[8] Y. Tian et al., “Identifying Software Changes that Induce Vulnerabilities,” in Proc. ICSE, 2012.
[9] J. Jiang et al., “A Survey on Large Language Models for Code Generation,” arXiv:2406.00515, 2024.
[10] L. Chen et al., “Evaluating Large Language Models in Code Generation Tasks,” arXiv:2408.16498, 2024.