AI-Driven Automated Code Review and Debugging Using Large Language Models

Chandan G K[1] Dr. Basavaraj S Pol[2] Manasa K[3] Dr. Avinasha P S[4]
[1]M.Tech Student [2]HOD and Associate Professor [3]Assistant Professor [4]Associate Professor
[1,2]Department of Computer Science and Engineering [3]Department of AIML (Artificial Intelligence and Machine Learning) [4]Department of Mechanical Engineering
[1,2]R L Jalappa Institute of Technology, DB pura, Bengaluru, India [3]BMS Institute of Technology & Management, Bengaluru, India [4]Cambridge Institute of Technology – North Campus, Bengaluru, India

Abstract — Software systems are growing in scale and complexity, and ensuring their quality through manual code review and debugging has become increasingly time-consuming, inconsistent, and dependent on the availability of experienced reviewers. This paper presents an AI-Powered Automated Code Review and Code Debugger, referred to as BugFix.AI, which combines static code analysis, machine learning, and large language model (LLM)-based reasoning to detect and correct programming errors with minimal human intervention. The proposed system parses submitted source code, classifies defects such as syntax errors, logical flaws, performance bottlenecks, and non-standard coding practices, and generates context-aware fixes together with human-readable explanations. Suggested corrections are validated through sandboxed execution before being presented to the user, ensuring that no new errors are introduced. The system was implemented using a React.js front end, a Node.js/REST API back end, and GPT-4o for reasoning, and was evaluated on a curated dataset of code samples spanning multiple programming languages. Experimental results show that the system achieves an overall bug-detection F1-score above 74%, reduces manual review effort by a substantial margin, and was rated helpful by a majority of surveyed users. The findings indicate that AI-driven automated review and debugging tools can meaningfully improve developer productivity and code quality while remaining a complement to, rather than a replacement for, human oversight.

Keywords: Artificial Intelligence, Automated Code Review, Code Debugging, Large Language Models, GPT-4o, Natural Language Processing, Static Analysis, Software Quality, Automated Testing

I. Introduction

Source code is at the core of nearly every modern technology, from healthcare and finance to autonomous systems and communication platforms. As applications grow larger and more interconnected, keeping code correct, readable, and maintainable becomes an increasingly demanding task. Traditionally, this responsibility has rested on developers and senior engineers who manually review changes and trace the causes of defects. Manual review is slow, subject to fatigue and oversight, and difficult to keep consistent across large teams, particularly under the rapid iteration cycles common to agile and DevOps practices.

Advances in artificial intelligence, and in large language models in particular, have opened new possibilities for automating parts of this workflow. Unlike traditional rule-based linters, which can only flag violations of predefined patterns, modern AI systems can reason about the intent behind a piece of code, infer likely causes of failure, and propose context-appropriate corrections along with natural-language explanations. This paper describes the design and evaluation of BugFix.AI, a system that unifies static analysis, machine learning-based error classification, and LLM-driven fix generation into a single automated code review and debugging pipeline.

The remainder of this paper is organized as follows. Section II reviews related work in static analysis and AI-assisted code review. Section III states the problem addressed by this work. Section IV describes the proposed system architecture. Section V details the methodology, including dataset preparation and model integration. Section VI discusses implementation. Section VII presents experimental results, and Section VIII concludes the paper with directions for future work.

II. Related Work

Traditional static analysis tools such as ESLint, Pylint, and SonarQube detect syntax errors and style violations by matching code against fixed rule sets. These tools are fast and require no execution of the code, but they cannot reason about program intent and therefore miss deeper logical or semantic defects [1], [2].

Machine-learning-based approaches, such as DeepCode and CodeGuru, improve on rule-based checking by learning error patterns from large annotated code corpora, allowing them to generalize to previously unseen bug types [3]. More recently, transformer-based large language models such as GPT-4o, CodeBERT, and Mistral have demonstrated an ability to understand code semantics and produce context-aware suggestions rather than isolated rule violations [4], [5].

Studies evaluating LLM-based review tools in production settings report mixed outcomes: while automated comments are frequently acted upon and defect detection improves, review latency can increase because developers must interpret and validate machine-generated suggestions [6], [7]. Other work proposes human-in-the-loop frameworks in which LLM output is treated as a recommendation rather than an automatic patch, and multi-agent pipelines that separate the tasks of hypothesis generation, verification, and explanation to reduce false positives [8], [9]. These findings motivate the design choices adopted in BugFix.AI, particularly its emphasis on sandboxed validation and human-readable explanations before any fix is applied.

III. Problem Statement

Ensuring high-quality, defect-free code remains difficult under real-world time constraints, limited access to experienced reviewers, and the growing scale of modern codebases. Manual review is labor-intensive and does not scale well for students, individual developers, or small teams. Beginner programmers in particular often struggle to identify

the root cause of a logical error or to apply established best practices without real-time, explanatory feedback. Existing static analysis tools stop at flagging an issue and rarely propose a validated fix, leaving the remaining diagnostic and corrective work to the developer. This paper addresses the resulting gap by developing an AI-powered system capable of detecting, explaining, and automatically correcting a broad range of code defects across multiple programming languages.

IV. Proposed System Architecture

BugFix.AI follows a layered, modular architecture consisting of a presentation layer, a processing layer, and an AI reasoning layer, connected through secure REST APIs. This separation of concerns allows individual components to be scaled, replaced, or extended without disrupting the rest of the system.

A. Code Parsing and Tokenization Module

Submitted source code is normalized and tokenized according to the grammar of its programming language, producing a structured representation such as an abstract syntax tree. This step exposes loops, conditionals, function boundaries, and object definitions that later modules use to reason about program structure rather than raw text.

B. Error Classification and Bug Detection Engine

Parsed code is analyzed to classify issues into categories such as syntax errors, logical bugs, performance concerns, and style violations. Classification combines deterministic rule checks with pattern recognition trained on labeled examples, allowing the system to catch both well-known and previously unseen defect types.

C. Fix Recommendation Engine

For each classified defect, the fix recommendation engine queries a large language model with a structured, chain-of-thought prompt that requests the defect category, severity, root cause, a corrected code snippet, and a plain-language explanation. Fixes range from small syntax corrections to broader restructuring, and are always returned together with a rationale.

D. Execution and Validation Engine

Proposed fixes are executed inside an isolated sandbox to confirm that they compile or run correctly and do not introduce new failures. Only fixes that pass this validation step are surfaced to the user as accepted suggestions; unresolved cases are flagged for manual review.

E. User Interface Module

A browser-based editor presents the submitted code, highlighted issues, and suggested corrections side by side, allowing the developer to accept, reject, or modify each fix. Optional automated email reports summarize the review for asynchronous workflows such as continuous integration pipelines.

V. Methodology

A. Dataset

Evaluation used a curated collection of code samples combining hand-crafted defect examples with samples drawn from public sources such as GitHub, CodeNet, and CodeSearchNet, covering Python, JavaScript, Java, and C/C++. Each sample was labeled by category of defect, severity, and an accepted correction, and the corpus was split into training, validation, and test partitions.

B. Preprocessing

Source code was normalized for whitespace, encoding, and comment style, then tokenized and converted into an abstract syntax tree to capture structural relationships between code elements. Embedding-based feature extraction was subsequently used to represent code numerically for pattern-based defect detection.

C. Model Integration and Prompt Engineering

Fix generation relies on GPT-4o accessed through a structured API call. A chain-of-thought system prompt instructs the model to identify the defect category, assign a severity level, explain the underlying cause, and produce both a corrected snippet and a best-practice recommendation. This structured prompting strategy was found to meaningfully improve correctness compared with unstructured prompting.

VI. Implementation

The system was implemented as a full-stack web application. The front end uses React.js with a Monaco-based in-browser code editor supporting syntax highlighting for multiple languages. The back end is built on Node.js and Express, exposing REST endpoints that coordinate static analysis, model invocation, and sandboxed execution. Language-specific linters (ESLint for JavaScript, Pylint for Python) perform an initial static pass before results and code context are forwarded to GPT-4o. Continuous integration is supported through GitHub Actions, which can trigger an automated review whenever new code is pushed to a repository and deliver a summary report by email.

VII. Results and Analysis

The system was evaluated on a held-out test set of 500 labeled code samples (250 defective, 250 clean) spanning Python, JavaScript, and Java. Table I summarizes detection performance by language.

Language	Prec.	Recall	F1
Python	79.2%	76.8%	78.0%
JavaScript	76.5%	74.1%	75.3%
Java	73.8%	71.4%	72.6%
C/C++	69.2%	66.7%	67.9%
Overall	75.3%	72.9%	74.1%

Table I: Bug Detection Accuracy by Language

Structured, chain-of-thought prompting improved overall correctness to 74.3%, compared with 68.5% for unstructured prompting of the same underlying model. Response latency remained under three seconds for files under 200 lines of code and under ten seconds for files up to

1,000 lines, which is well within the range needed for interactive use during development.

A four-week pilot with a small group of developers showed that time spent on manual bug detection and correction fell substantially after adopting the tool, most notably for repetitive or well-understood defect categories such as missing braces, unmatched parentheses, and off-by-one loop errors. A follow-up survey of 82 participants, spanning students and professional developers, found that 77% rated the system "helpful" or "very helpful," and a majority reported improved understanding of the underlying cause of their bugs rather than simply receiving a corrected file. Consistent with prior studies on LLM-assisted review, participants noted that unclear or overly broad suggestions occasionally required manual verification, reinforcing the design decision to keep a human-in-the-loop confirmation step before any fix is applied.

VIII. Conclusion and Future Scope

This paper presented BugFix.AI, an AI-powered automated code review and debugging system that combines static analysis, machine learning-based defect classification, and large language model reasoning within a sandboxed validation pipeline. Experimental results demonstrate that the system reliably detects and corrects a wide range of programming errors, reduces manual debugging effort, and is well received by both student and professional developers, while remaining most effective when used to complement rather than replace human review.

Future work will extend language coverage beyond the languages evaluated here, incorporate real-time collaborative review, add version-control-aware analysis for pull requests, and explore lightweight offline models for basic syntax checking without network access. Integrating the system directly into IDEs and CI/CD pipelines, along with a learning-analytics dashboard that tracks recurring error patterns over time, is expected to further embed automated review into everyday development workflows.

Acknowledgment

The authors thank the Department of Computer Science and Engineering, R. L. Jalappa Institute of Technology, Doddaballapura; the Department of Artificial Intelligence and Machine Learning, BMS Institute of Technology & Management, Bengaluru; and the Department of Mechanical Engineering, Cambridge Institute of Technology – North Campus, Bengaluru, for the resources and support provided during this work.

References

[1] P. Louridas, "Static code analysis," IEEE Software, vol. 23, no. 4, pp. 58–61, 2006.
[2] R. Tufano, A. Martin-Lopez, and A. Tayeb, "Deep learning-based code reviews: A paradigm shift or a double-edged sword?," 2024.
[3] Z. Li, S. Lu, and D. Guo, "Automating code review activities by large-scale pre-training," in Proc. ESEC/FSE, Singapore, 2022, pp. 1035–1047.
[4] U. Cihan, A. Icoz, V. Haratian, and E. Tuzun, "Evaluating large language models for code review," arXiv:2505.20206, 2025.
[5] S. Kang, B. Chen, and S. Yoo, "Explainable automated debugging via large language model-driven scientific debugging," 2023.
[6] U. Cihan and V. Haratian, "Automated code review in practice," 2024.
[7] J. Lu, L. Jiang, and X. Li, "Towards practical defect-focused automated code review," in Proc. 42nd ICML, PMLR 267, Vancouver, 2025.
[8] M. A. Haider, A. B. Mostofa, and S. S. B. Mosaddek, "Prompting and fine-tuning large language models for automated code review comment generation," arXiv:2411.10129, 2024.
[9] W. Yang, H. Wang, and Z. Liu, "COAST: Enhancing the code debugging ability of LLMs through communicative agent-based data synthesis," Assoc. for Computational Linguistics, 2025.
[10] G. Fan, X. Xie, and X. Zheng, "Static code analysis in the AI era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents," 2023.

