Autonomous Code Reviewer Using LLMs and Static Analysis

[1]Dr. P.V.S. Sarma,[2]Vadalasetty Jyothi,[3]Singiresu Rajesh,[4]Shaik Kaleshavali

[1]Associate Professor, Dept of Computer Science and Engineering, St. Ann's College of Engineering and Technology, Chirala-523187, India.

[2,3,4]B. Tech Student, Dept of Computer Science and Engineering, St. Ann's College of Engineering and Technology, Chirala-523187, India.

ABSTRACT

It is an intelligent, multi-language code analysis system designed to automatically evaluate individual code files as well as complete software project folders uploaded in ZIP format. The system extracts and analyzes every source code file inside the ZIP, regardless of the programming language, and performs deep code understanding using Large Language Models (LLMs) combined with static code analysis techniques.The LLM component interprets the code like a human reviewer—understanding logic, structure, flow, intent, design patterns, and readability—while detecting bad practices, inefficiencies, and recommending optimized, cleaner versions of the code. Simultaneously, the static analysis engine identifies syntax errors, bug patterns, memory risks, null-pointer issues, type mismatches, unreachable code, vulnerabilities, and code smells. By merging the strengths of both approaches, the system produces a comprehensive and context-aware review report covering all files within the project. This includes detailed bug detection, security analysis, and performance suggestions, along with an improved, efficient rewrite of problematic code sections. The ability to analyze entire project ZIP files makes the system scalable, practical, and highly useful for real-world development workflows. Overall, the system significantly reduces manual review effort and helps developers deliver secure, optimized, and maintainable code across multiple languages.

KEYWORDS: ***Autonomous Code Review, Large Language Models, Static Analysis, Artificial Intelligence, Software Quality, Bug Detection, Code Optimization.***

NTRODUCTION

The rapid expansion of software applications across industries has significantly increased the demand for high-quality, reliable, and secure code. As systems grow more complex, maintaining code quality becomes a challenging task for developers and organizations. Traditional manual code review processes require skilled developers and consume considerable time and effort, making them inefficient for large-scale projects. Moreover, human reviewers may overlook critical issues due to fatigue or limited context understanding.

To address these challenges, automated code analysis techniques have been introduced to assist developers in identifying errors and improving code quality. Static analysis tools can detect

syntax errors, vulnerabilities, and code smells, but they rely on predefined rules and lack contextual awareness. Recently, advancements in Artificial Intelligence, particularly Large Language Models (LLMs), have enabled machines to understand code semantics and provide intelligent suggestions. These models can interpret logic, detect inefficiencies, and recommend optimized solutions. However, LLMs alone may miss low-level technical flaws that static tools can identify. Therefore, integrating LLMs with static analysis offers a powerful approach for building intelligent and comprehensive code review systems.

RELATED WORK

Several research efforts have focused on improving automated code analysis using traditional and modern techniques. Early tools such as static analyzers were designed to identify syntax errors, memory leaks, and security vulnerabilities based on predefined rules. These tools proved useful but often generated a high number of false positives and lacked the ability to understand code context. Later, machine learning approaches were introduced to enhance bug detection and pattern recognition in source code. Researchers explored models trained on large codebases to predict defects and recommend fixes.

With the emergence of deep learning, transformer-based architectures have shown promising results in code understanding and generation tasks. Systems like AI-based code assistants can now provide suggestions, detect logical errors, and improve readability. However, many of these systems focus either on static analysis or AI-based interpretation, but not both. Some hybrid approaches have attempted to combine these techniques, yet they remain limited in scalability and multi-language support. Additionally, most existing systems analyze individual files rather than entire projects. This highlights the need for a more integrated and scalable solution for comprehensive code review.

LITERATURE SURVEY

The literature on automated code review reveals significant progress in both static analysis and AI-driven techniques. Studies on static code analysis emphasize its effectiveness in detecting low-level issues such as syntax errors, null pointer exceptions, and security vulnerabilities. However, these methods are limited by their rule-based nature and inability to interpret developer intent. On the other hand, recent research on Large Language Models demonstrates their capability to understand programming languages and generate human-like explanations.

These models have been applied in code summarization, bug detection, and automated code generation tasks. Researchers have also explored combining AI models with traditional tools to improve accuracy and reduce false positives. Some works highlight the importance of context-aware analysis for identifying complex logical flaws in software systems.

Despite these advancements, challenges remain in analyzing large-scale multi-file projects and maintaining consistency across different programming languages. Furthermore, existing approaches often lack real-time feedback and scalability. This survey indicates a gap in developing a unified system that integrates deep learning with static analysis for efficient and comprehensive code review.

EXISTING METHOD

Traditional code review methods primarily rely on manual inspection performed by experienced developers to ensure code quality and correctness. This process is time-consuming and becomes inefficient when dealing with large-scale software projects. Static code analysis tools were introduced to automate error detection by scanning source code without execution. These tools can identify syntax errors, memory leaks, and common vulnerabilities using predefined rules. However, they often generate a high number of false positives, which can mislead developers and increase debugging effort. Additionally, static tools lack the ability to understand program context, logic, and developer intent. Some machine learning-based approaches have been proposed to enhance defect prediction and pattern recognition in code. Despite this, they are usually limited to specific programming languages or datasets. Existing systems also tend to analyze individual files rather than entire project structures. As a result, they fail to provide a comprehensive and context-aware review of complex software systems.

PROPOSED METHOD

The proposed system introduces an intelligent and integrated approach for automated code review by combining static analysis with Large Language Model (LLM)-based understanding. Users can upload individual code files or complete project folders in ZIP format, enabling scalable and real-world applicability. The system first performs preprocessing to extract, filter, and normalize source code files across multiple programming languages. Static analysis techniques are then applied to detect low-level issues such as syntax errors, vulnerabilities, and code smells. Simultaneously, the LLM module analyzes the code to understand its logic, structure, and intent, providing human-like insights. It evaluates readability, maintainability, and efficiency while suggesting optimized and refactored code. The outputs from both modules are intelligently merged to eliminate redundancy and improve accuracy. Issues are prioritized based on severity, ensuring critical problems are addressed first. The system generates a unified, detailed report with explanations, performance improvements, and security recommendations. This approach significantly reduces manual effort and delivers a comprehensive, context-aware code review solution.

SYSTEM ARCHITECTURE

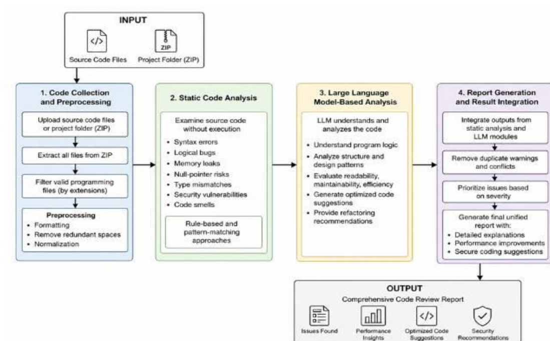

Fig 1: Block Diagram

The implementation of the Autonomous Code Reviewer involves multiple components working together to analyze and evaluate source code. The backend of the system is developed using Python and Flask, which handles file uploads, code processing, and integration with analysis modules. The frontend interface is built using HTML, CSS, and JavaScript to provide a user-friendly platform for uploading code files and viewing analysis results. When a user uploads a ZIP file

containing project source code, the system extracts all files and identifies programming languages using a language detection module. The static analysis module scans each file and identifies potential bugs, syntax errors, and security vulnerabilities using predefined rules. The LLM-based analysis module then processes the code using transformer-based models capable of understanding programming language semantics. This module evaluates the overall quality of the code and suggests improvements. The results from both modules are combined and presented to the user as a detailed code review report containing detected issues, severity levels, explanations, and optimized code suggestions.

METHODOLOGY DESRIPTION

Code Collection and Preprocessing

Users upload source code files or project folders in ZIP format. The system extracts all files and filters valid programming files based on extensions. Preprocessing techniques such as formatting, removal of redundant spaces, and normalization are applied to prepare the code for analysis [1].

Static Code Analysis

Static analysis techniques are used to examine the source code without execution. This method identifies syntax errors, logical bugs, memory leaks, null-pointer risks, type mismatches, security vulnerabilities, and code smells using rule-based and pattern-matching approaches [2].

Large Language Model-Based Analysis

The Large Language Model (LLM) module analyzes the code to understand program logic, structure, design patterns, and developer intent. It evaluates readability, maintainability, and efficiency, and generates optimized code suggestions and refactoring recommendations [3].

Report Generation and Result Integration

The outputs from static analysis and LLM modules are integrated to form a unified review report. Duplicate warnings are removed, and issues are prioritized based on severity. The final report presents detailed explanations, performance improvements, and secure coding suggestions in a user-friendly format [4].

RESULTS AND DISCUSSION

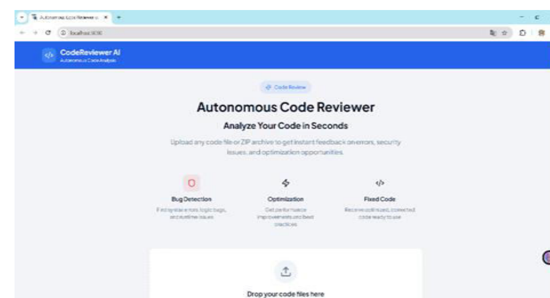

Fig 1: Home Page

Fig 1 shows Autonomous Code Reviewer system. It provides a simple and user-friendly interface for developers to upload code files or ZIP project folders for analysis. The page highlights key features such as bug detection, code optimization, and fixed code suggestions. Users can easily upload their files using the drag-and-drop upload section.

After uploading, the system analyzes the code and generates an automated review report with errors and improvements. The interface is designed to make code review

faster and more efficient. It helps developers improve code quality and follow better programming practices.

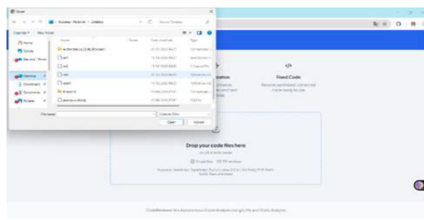

Fig 2: upload page

Figure 2 shows the file upload interface of the Autonomous Code Reviewer system. This section allows users to upload individual source code files or complete project folders in ZIP format. Users can either drag and drop files into the upload area or select them manually from their device. Once the file is uploaded, the system automatically processes the code for analysis. The uploaded files are then prepared for static analysis and LLM-based code understanding. This feature enables quick and convenient submission of code for automated review. It ensures that developers can easily analyze their projects and receive instant feedback.

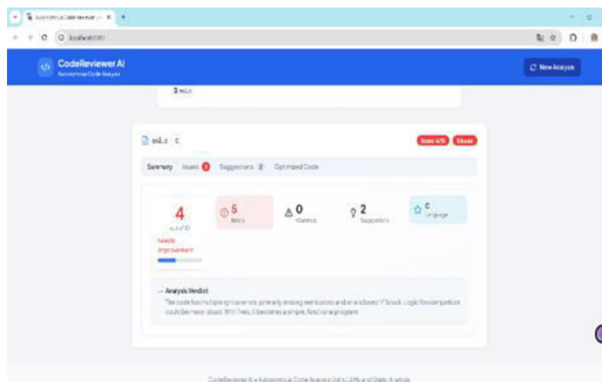

Fig 3: Code Analysis Results

Figure 3 shows the code analysis result page of the Autonomous Code Reviewer system. After uploading the source code file, the system analyzes it and displays the results in a structured format. The page provides a code quality score, number of errors, warnings, and suggestions identified in the program. It also shows the programming language used and an overall analysis verdict describing the issues found in the code. The system highlights problems such as missing syntax elements or logical errors.

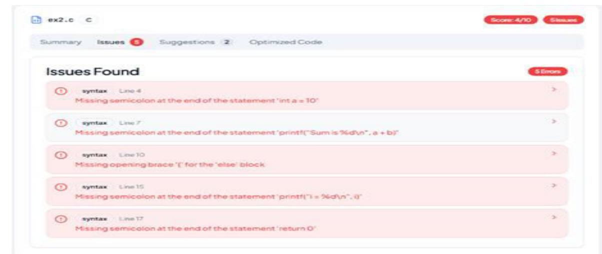

Fig 4: Issues Present in Code File

Figure 4 shows the Issues section of the Autonomous Code Reviewer system. This section displays the errors and problems identified in the uploaded source code during analysis. The system highlights issues such as syntax errors, missing statements, logical mistakes.

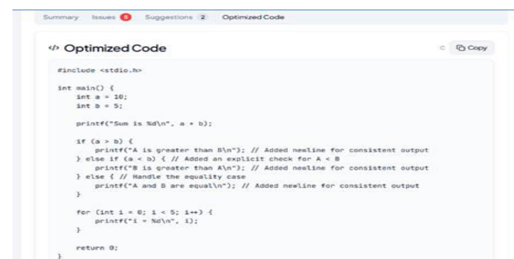

Fig 5: Optimized Code

Figure 5 shows the Optimized Code section of the Autonomous Code Reviewer system. This section provides an improved version of the uploaded source code after analysis. The system corrects syntax errors and suggests better coding practices to improve readability and performance. It rewrites inefficient or incorrect parts of the program with optimized logic. Developers can compare the original code with optimization.

CONCLUSION

Our project presents an autonomous code reviewer that combines Large Language Models with static analysis to automatically evaluate source code. It accurately detects bugs, errors, and security issues while providing meaningful suggestions and optimized code. The system reduces manual review effort and helps developers produce secure, efficient, and high-quality software.

FUTURE SCOPE

In the future, the system can be extended to support more programming languages and frameworks. Real-time integration with IDEs and version control platforms like GitHub can enable continuous code review during development.

REFERENCES

1. Harini, D. P. (2012). codes: A Collaborative spam Detection system with a novel E-mail abstraction scheme. *IDSR Journal of Engineering*.

2. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," ACM Computing Surveys, vol. 51, no. 4, 2018.

3. V. Raychev, M. Vechev, and E. Yahav, "Probabilistic Model for Code with Decision Trees," ACM SIGPLAN Conference on Programming Language Design and Implementation, 2016.

4. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" International Conference on Software Engineering (ICSE), 2013.

5. J. Sadowski, E. Aftandilian, and A. Eagle, "Lessons from Building Static Analysis Tools at Google," Communications of the ACM, vol. 61, no. 4, 2018.

6. A. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," IEEE Transactions on Software Engineering, 2019.

7. Y. Liu, S. Li, Y. Zhang, and H. Wang, "Deep Learning-Based Code Defect Prediction," IEEE Access, vol. 7, 2019.

8. M. Pradel and K. Sen, "DeepBugs: A Learning Approach to Name-Based Bug Detection," Proceedings of the ACM on Programming Languages, 2018.

9. A. Vaswani et al., "Attention Is All You Need," Advances in Neural Information Processing Systems (NeurIPS), 2017.

10. D. Guo et al., "GraphCodeBERT: Pre-Training Code Representations with Data Flow," International Conference on Learning Representations (ICLR), 2021.

11. T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," arXiv preprint, 2013.

12. S. Panichella, G. Bavota, M. Di Penta, and R. Oliveto, "How Developers' Experience and Expertise Influence Code Review Quality," IEEE Transactions on Software Engineering, 2018.