Grenze International Journal of Engineering and Technology, January Issue

Code Reviewer: Intelligent Code Review System using AI APIs for Feedback, Rating, and Refactoring

Mrs. G G Sreeja[1], Karthik S[2], Dr. N. Ashokkumar[3], Vishal Kumar S[4], Tamilkumaran N[5] and Akshayan K[6]
[1-2,4-6]Department of CCE, Sri Eshwar College of Engineering, Coimbatore, India
Email: sreeja.gg@sece.ac.in, karthik.saravanan3011@gmail.com, vishalteejay007@gmail.com, tamilkumaran1494@gmail.com, marvelakshayan@gmail.com
[3]Department of ECE, Mohan Babu University, Tirupati, India
Email: ashoknoc@gmail.com

***Abstract*— Code quality is central to the efficiency, maintainability, and scalability of software systems. Traditional manual code reviews, as effective as they are, are resource-inefficient, inconsistent, and prone to human error. Code Reviewer, a multi-language code review tool backed by AI to automate and enhance the review process, is presented in this paper. The software integrates multiple publicly accessible large language models (LLMs) like Gemini, DeepSeek, and Claude to review code, suggest improvement, provide ratings for quality, and produce optimized code. Developers can submit code in various programming languages through an easy web interface and receive model-specific real-time feedback. Comparative analysis of AI output for the selection of the most suitable result is also supported by the system. Deployed using React.js at the frontend and Node.js at the backend, the solution employs AI APIs to provide an affordable, smart, and automated code review platform.**

***Keywords*— Code review, artificial intelligence, large language models, Gemini API, DeepSeek, code optimization, software quality, multi-language support, real-time feedback, AI-driven development tools.**

I. INTRODUCTION

For high-quality code to be obtained in current software development, reliability, maintainability, security, and scalability of software must be obtained [1]. Code quality is directly accountable for defect density, system performance, ease of future modification, and long-term existence of a project. Because of this, code review has become a standard practice in software engineering processes, often mandated in professional environments to guarantee coding standards are maintained and defects are minimized to zero prior to release. Peer or senior developers' manual code review is normally applied in the traditional code review. This process involves code reviewing for correctness, style conformance, optimization, and security risk [2]. Manual reviews, as efficient as they might be theoretically, are subject to a number of drawbacks. Firstly, such processes are time-consuming, particularly in large code bases or for high-intensity agile development environments. Second, quality review is typically unpredictable due to variations in reviewer skill level, human bias, and workload limitations [3]. Human reviews are often language-specific because reviewers are focus on programming languages where they have experience, thereby having limiting in cross-language accessibility. Such limitations render the correctness of the review and difficulties in scalability to ensure in changing development lifecycles.

Grenze ID: 01.GIJET.12.1.528_1

New advanced features in artificial intelligence (AI), mostly large language models (LLMs), have helped to have a gateway to new possibilities for solving these kinds of problems. Gemini, DeepSeek, and Claude are the examples of models that have been able to analyze natural language, understand programming concepts, and suggest code that shows human-like reasoning's [4]. Educated on more technology and various programming compilations, these models have the ability to find logical bugs, optimise codes for the best performance enhancement, and help to have rich commentary for various programming languages. This feature makes LLMs effective tools for automating code review's knowledge-intensive process.

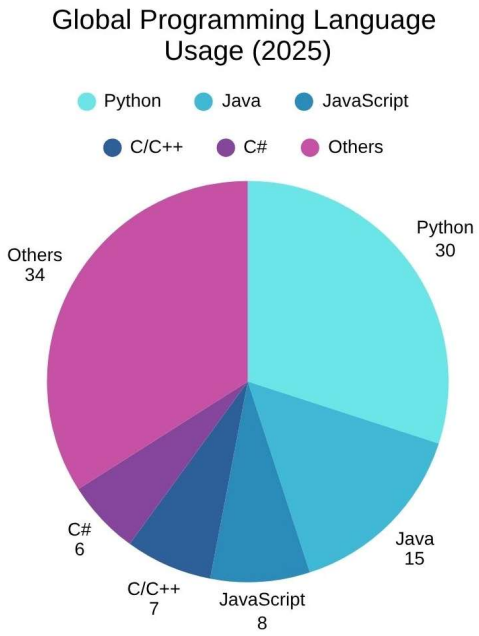

Figure 1: Popular Language Usage

Yet, while promising, current AI-based code review tools are confronted by two major obstacles: limited access through subscription pricing and inadequate integration of multiple models to enable comparative analysis. This opens up the need for a low-cost, multi-model, and multi-language code review system that can deliver in-depth, impartial feedback. To fill this gap, we introduce Code Reviewer, an AI-facilitated, multi-language code review tool that uses LLM APIs such as Gemini, DeepSeek, and Claude. The tool allows developers to upload code using a secure, browser-based interface and obtain real-time feedback on code quality, optimization recommendations, maintainability scores, and refactored versions of code. One unique feature of the system is that it provides review outputs of different AI models simultaneously so developers can evaluate the outcomes and choose the best contextually related suggestion. The architecture of the system includes a React.js client for user interface and a Node.js server for API interaction and processing. Review requests are sent securely over HTTPS, and results are aggregated and shown in an organized, developer-targeted fashion. By combining several AI models into one platform, Code Reviewer increases developer productivity, minimizes the need for manual peer reviews, speeds up development cycles, and encourages better coding standards.

II. Background Work

Software development advancements have increasingly focused on enhancing code quality, maintainability, and automation of workflows. Manually reviewing codes by human reviewers has conventionally been the conventional way to attain these goals. This process, however, is usually time-consuming, susceptible to missing issues, and unreliable because of inconsistent reviewer skills and contextual knowledge [10]. As the need to accelerate release cycles grows, research has moved towards implementing artificial intelligence into the review process. Current studies have been using machine learning and deep learning models from large-scale code bases to automate review processes. These models learn from past reviewer comments and code changes to detect

patterns of quality improvement. Some seek to duplicate reviewer comments or introduce changes automatically, and others point out recurring problems based on learned heuristics [11]-[13]. Although showing promising performance, these approaches tend to depend on large datasets, complex training pipelines, and a lot of computational resources, thus constraining their usability for small teams and individual developers. The existence of big language models (LLMs) like Gemini, DeepSeek, and Claude has provided a more accessible alternative. Trained on large programming corpora beforehand, these models can perceive and produce code-related information, such as suggestions, refactoring, and explanations, without domain-specific training. They also provide wide multi-language support, allowing analysis across languages like C, C++, Java, and Python, which breaks the single-language restriction of previous systems. Another difficulty in previous methods is guaranteeing the reliability and security of automatic suggestions. Without sufficient context or validation, such outputs can potentially degrade functionality or introduce defects. By integrating multiple LLMs and enabling users to compare their outputs, the proposed system enhances transparency and provides developers with greater control in selecting the most suitable recommendations. Overall, earlier AI-assisted code review research has laid an important foundation, yet accessibility, scalability, and usability challenges remain. The Code Reviewer solution solves these problems through a API-based, multi-language platform that provides sophisticated review functionality without requiring specific hardware or huge training sets.

III. Software Requirements for App Implementation

Creation of an intelligent, AI-based code review platform like Code Reviewer demands the integration of various software tools, APIs, and frameworks for providing a smooth, secure, and scalable experience to developers. From frontend interface design to backend API integration and AI processing, every piece of software works to offer the complete functionality. Figure 2 shows the overall architecture and data flow for the platform, including user input, API interaction, and the presentation of response from large language models (LLMs).

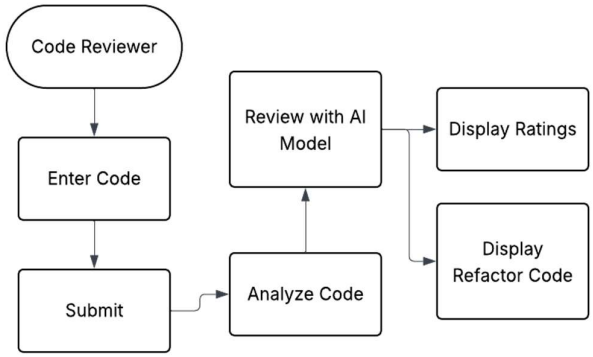

Figure 2: Flowchart for application

The process of development is performed with a stable Integrated Development Environment (IDE) for the frontend and backend deployment. Visual Studio Code is utilized to develop frontend using React, and Node.js with Express handles backend API routing and asynchronous interaction with AI services. They provide a strong debugging mechanism, support for extensions, and cross-platform compatibility, thus easing web application development. React.js is selected due to its component-based structure and rich ecosystem, enabling the production of an interactive, responsive interface. Further styling and layout control are enabled by libraries like Tailwind CSS or Material UI. A native code editor, written with CodeMirror or Monaco Editor, provides the ability to write and view code with syntax highlighting and auto-formatting options.

The backend server, which is implemented in Node.js and Express, is the middleware that links the frontend to the LLM APIs. It prepares prompts, detects errors, authenticates (if need be), and processes the response from the AI. All communications are done over HTTPS for confidentiality and integrity. The major functionality of the platform comes from LLM APIs like Google Gemini, DeepSeek, and Claude. These APIs review submitted code and provide suggestions for improvement, quality scores, and refactored code alternatives. Responses are returned in a parsed, structured format and can be used for real-time result comparison by users. Error-handling procedures and fallbacks are adopted in order to handle API rate limits or lag. Versioning is handled by Git, and repository hosting is provided using GitHub, facilitating collaboration and rollback while developing. The present prototype runs without a specialized database, since permanent user accounts and history are not a

necessity. But scalable solutions like Firebase or MongoDB Atlas can be introduced in subsequent versions to save user preferences, previous reviews, or analytical information. If authentication becomes a requirement, secure solutions like Firebase Authentication or OAuth2 can be used. Testing is done with Jest for frontend unit tests and Postman for backend API testing. Deployments can be Vercel, Render, or Netlify for hosting the frontend and Railway, Render, or Glitch for the backend services. Security practices include CORS policies, input sanitization, API rate limiting, and safe storage of API keys in environment variables to avoid exposure. User interface design is prototyped with tools like Figma or Adobe XD to provide intuitive navigation and ease of use. The architecture of the system follows a lightweight, modular approach that hosts a responsive React-based UI, scalable backend services, and loosely coupled API integrations. The final platform that is produced is a pragmatic, cost-efficient solution for automated and smart code reviews, obviating the requirement for proprietary tools or costly infrastructure.

VI. Methodology

The evolution of the Code Reviewer platform is an organized and iterative process starting with a deep understanding of user needs, project goals, and the functional aspects of the existing AI technologies. The first phase was requirement analysis, which was done by thoroughly studying current manual code review processes and determining key deficiencies like inconsistency in comments, restricted language support, and response latency. These were condensed into essential platform attributes, such as real-time AI-based analysis, multi-language support, concurrent integration of multiple large language models (LLMs), and a single interface for side-by-side comparison of outputs. After the feature set was determined, the second phase was all about interface design and user experience planning. Wireframes and interactive mockups were developed using Figma to create a clean, developer-friendly workflow.

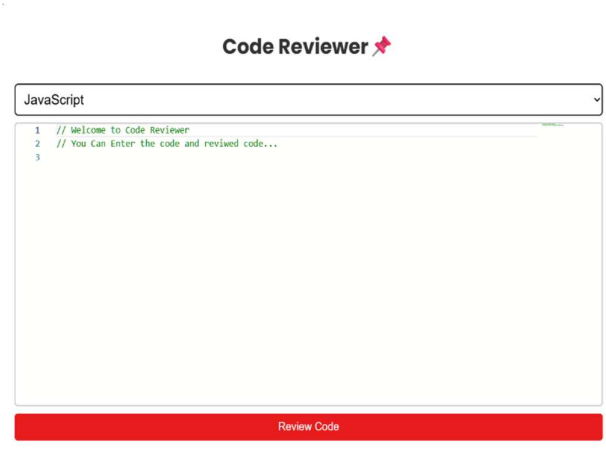

Figure 3. Sample User Interface

The interface was formulated to support pasting or uploading code, indicating the programming language, and immediate AI feedback. Most importance was given to usability to provide a small learning curve and smooth navigation for inexperienced and experienced programmers alike. The frontend implementation was performed using React.js because of its component-based structure, fast virtual DOM rendering, and extensive ecosystem to construct responsive, dynamic applications. A code editor element, built with CodeMirror, was added to enable syntax highlighting, automatic indenting, and language-specific formatting to provide a clear and easy-to-use editing experience. The backend component is based on Node.js and Express.js, selected for their event-driven, non-blocking design that allows high-performance, scalable request processing. This backend is also tasked with accepting user code submissions, preprocessing prompts for AI models, and handling asynchronous API calls to several LLMs—specifically Google Gemini, DeepSeek, and Claude. When code is submitted, the backend sends the code simultaneously to all supported AI models, normalizes their outputs, and pushes the consolidated results to the frontend for presenting in a single view. The design was purposefully made modular so that it would be straightforward to add future support for other AI models or features without impacting current components. Security and stability were also kept top of mind during development. Sensitive API credentials are all stored in secured environment variables, and HTTPS communication is enforced by the platform to secure confidentiality

and integrity of data in transit. Other safeguards are input sanitization to avoid injection attacks, rate limiting to manage excessive API usage, and comprehensive logging to monitor system health and debug Testing was performed continuously during the development cycle.

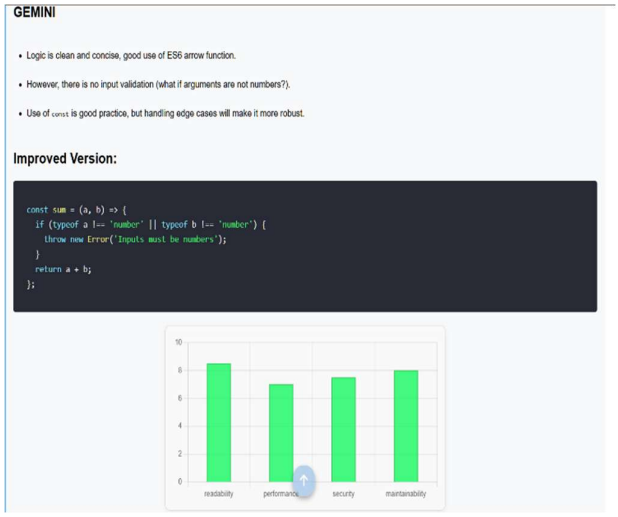

Figure 4: Sample Home page

Unit tests confirmed the operation of individual frontend elements and backend operations, while integration testing guaranteed error-free communication among layers of the system. This was followed by user acceptance testing, where student developers and faculty mentors assessed the application for the quality of feedback, usability, and performance under real-world scenarios. The last deployment used Vercel for the frontend and Render for the backend, which offered cloud-based scalability, low latency, and continuous integration pipelines to facilitate constant updates. The system's operational workflow starts with a user committing code through the web interface. The code is pushed to the backend, which at the same time sends it to the integrated LLMs. Every AI model examines the submission in isolation, producing formatted comments on areas like time complexity, memory optimization, code style compliance, bug identification, and best-practices suggestions. Where suitable, the AI can also provide refactored code snippets. The backend compiles these results into a single presentation, emphasizing overlapping feedback and distinct insight from each model. Users are able to view the grouped suggestions, compare improvements produced by AI, and implement the solution that suits their needs the best. This approach makes sure that the code reviewer is capable of providing quick, uniform, and multi-faceted feedback, closing the gap between manual code review and intelligent, automatic assessment. The resulting system not only shortens review time but also improves the quality of the recommendations by taking advantage of the strengths of collective AI systems. In addition, its module-based design enables future plug-ins of more models with ease to facilitate flexibility against future advances in AI. The project experience was more like developing an actual collaborative tool than a prototype. Every stage of testing provided us with a better understanding of how the system would actually be used by the students and the mentors. To watch it grow into a scalable, AI-powered reviewer was rewarding and motivating for the whole team.

V. LIMITATIONS

The suggested AI-powered code review system is by default reliant on third-party large language model (LLM) APIs for processing and delivering the feedback. Due to this, the functionality, availability, and consistency of the system depend on the operational uptime of such external services. Even temporary downtime, functional slowing down, or request caps by the providers could interrupt the responsiveness and availability of the platform. In addition, the system's performance is also limited by usage quotas of APIs, which can also cap the number of reviews to be processed per time frame, especially when demand is high. As much as the platform is made to accommodate a variety of programming languages, the accuracy, depth, and contextual response of feedback produced can differ across languages. This difference significantly relies on the amount and mix of each language's presence in the training sets of the combined AI models. For languages or platforms with minimal training exposure, the system can generate less inclusive reviews or overlook specific domain-specific

best practices and thus decrease the overall functionality for expert development activities. The platform is now optimized to work best on analyzing small- to medium-sized code pieces. However, when analyzing huge codebases or files with high dependency and logic depth, the computational expense and processing time may become significantly higher. This latency not only blocks code analysis speed, but also it may block development patterns where quick feedback is essential.

This limitation is only complicated by the absence of an integrated optimization process for managing large files. The most important limitation lies in the analytical nature of AI-driven suggestions. As much as the system can identify the area of possible improvement, bugs in code, or incompetence, the suggestions themselves are not always fully correct and may not be feasible in all cases. Differences in the logic, coding patterns, or project conditions can guide the difference between the suggestions and the planned functionality of the code. As a result, developers must think carefully and make sure to test each suggestion before integration to analyze consistency in alignment with project goals and conditions. Finally, as the system is combining suggestions from different AI models, differences in their training procedures and output styles may conclude in non-uniform outcomes. Although this variety has the ability to develop an extensive understanding of possible improvement, it may also include complexity in establishing which suggestion to follow. In the absence of a systematized ranking or screening procedure, the decision-making load would fall purely on the end user; possibly it will increase the difficulty in decision-making. To solve these kinds of problems, it will require future work integrating caching, prioritizing the methods, and flexible ranking procedures. With this implementation, the platform can grow into a more trustworthy, scalable, and smart code review tool. Overall, even though the system has great potential, its actual usage is limited by technical as well as restrictions from outside services.

VI. Future Scope

The growing use of artificial intelligence in software development points to the potential for tools such as Code Reviewer to become critical applications for improving code quality and optimizing development workflows. In the future, the system can be expanded by adding features such as real-time collaborative reviewing, user-adaptive learning modules, and ease of integration with version control platforms like GitHub. By tapping into information about a developer's programming history and code style, the platform would provide tailored recommendations, learning plans, and custom feedback and thus become an intelligent coding assistant. Gains in cutting-edge areas like Explainable AI (XAI) and natural language reasoning present possibilities to enhance transparency in automated feedback. Incorporating such technologies would allow the system to render elaborate explanations of its recommendations, leading to enhanced comprehension of programming principles. Further, adding support for multilingual source code as well as explanatory material would increase accessibility among non-English-speaking developers and educational institutions across the globe. With the development of large language models, the site's ability to evaluate efficiency, security, scalability, and correctness with respect to design principles will grow, allowing more thorough and industry-specific code reviews.

VII. Conclusion

The Code Reviewer platform is not just a piece of software—it's a fundamental change in how developers engage with code analysis and learning. It's more than just an AI-driven application, as it acts as a translator between human work and machine intelligence, enabling developers to create better, cleaner, and more efficient code. By streamlining sections of the otherwise manual and time-consuming review process, this platform boosts development cycles while facilitating ongoing learning through actionable feedback and code optimization. The use of multiple AI models provides diverse sets of suggestions and viewpoints, enabling the user to make informed choices when enhancing their work. This project is an example of how contemporary AI technologies can be employed to raise not just productivity but also knowledge in the area of software development. The system promotes optimal coding habits, exposes users to different logic streams, and accommodates a variety of languages, making it inclusive and versatile for both professionals and learners. Apart from its technical excellence, Code Reviewer illustrates a larger vision: one of intelligent systems augmenting human development—growth, creativity, and problem-solving. It encourages learning and sharing by allowing individuals to review, refine, and develop with each line of code they create. Code Reviewer is truly a step toward democratizing software quality, making intelligent code insights available to anyone, anywhere, in any situation, no matter the background or resources. Its worth is not only in code metrics of improvement but also in the confidence it inspires in its users, the education it creates, and the potential it realizes for cleaner, better, more meaningful software. As we continue to progress technologically, sites such as this illustrate how

effectively and ethically AI can be used to advance human potential—making development smarter, faster, and available to everyone. In the future, such a platform can further develop into a collaborative environment where developers, instructors, and AI systems are able to operate in tandem. By merging innovation with ease of access, Code Reviewer is a testament to how technology can not only enable the current generation of coders but also stimulate the next generation of problem-solvers.

REFERENCES

[1] C. B. de Souza, L. D. G. Silva, M. Aniche, et al., "Code Review Comprehension: A Cognitive Perspective," Empirical Software Engineering, vol. 27, no. 3, pp. 1–30, 2022. DOI: 10.1007/s10664-021-10001-1

[2] C. Bird, A. Bachmann, E. Aune, et al., "Fair and Balanced? Bias in Code Review," Proceedings of the 2015 37th International Conference on Software Engineering, pp. 499–510, IEEE, 2015. DOI: 10.1109/ICSE.2015.63

[3] A. Beller, S. McIntosh, and M. Zaidman, "Improving Code Review by Recommending Understandability Improvements," 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 67–78. DOI: 10.1109/SANER.2018.8330206

[4] M. Tufano, C. Watson, G. Bavota, et al., "An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," ACM Transactions on Software Engineering and Methodology, vol. 30, no. 4, pp. 1–29, 2021. DOI: 10.1145/3450304

[5] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to Represent Programs with Graphs," International Conference on Learning Representations (ICLR), 2018. [Online]. Available: https://arxiv.org/abs/1711.00740

[6] M. Pradel and K. Sen, "DeepBugs: A Learning Approach to Name-Based Bug Detection," Proc. ACM Program. Lang., vol. 2, OOPSLA, pp. 1–25, 2018. DOI: 10.1145/3276517

[7] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 10, pp. 1345–1359, Oct. 2010. DOI: 10.1109/TKDE.2009.191

[8] A. Vaswani, N. Shazeer, N. Parmar, et al., "Attention is All You Need," Advances in Neural Information Processing Systems, vol. 30, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[9] GitHub Copilot: Your AI Pair Programmer, GitHub, 2023. [Online]. Available: https://github.com/features/copilot

[10] Google AI, "Gemini API Developer Guide," Google Developer Portal, 2024. [Online]. Available: https://ai.google.dev/gemini

[11] OpenAI, "Understanding GPT-4 Capabilities," OpenAI Docs, 2024. [Online]. Available: https://openai.com/gpt-4

[12] DeepSeek, "DeepSeek Coder: Open-source LLM for Code Generation," 2024. [Online]. Available: https://github.com/deepseek-ai