AI-Powered Code Quality Analyzer and Error Fixer: An Intelligent Framework for Automated Code Review, Optimization, and Quality Assessment

Dr.P.Veeresh[1], M Jagadeesh[2], Saparay Deepak[3], Angadi Sai Yashwanth[4], V Rahul Kumar[5], Poojari Vivekananda Sai[6]

[1,2,3,4,5,6]*Department of Computer Science and Engineering, St. Johns College of Engineering and Technology, Yemmiganur-518301, India*

Abstract- Modern software development, it is of utmost importance to ensure the quality of the code to ascertain the reliability, maintenance, and quality service of the program. In most situations, and particularly among students and novice programmers, the challenge is to ensure the production of quality or optimized code due to minimum exposure to code quality standards and practice. In the traditional practice of code review, a programming expert is needed to enhance code quality, and the review of the code is always a lengthy and non-scalable end, among other limitations. However, to overcome the aforementioned challenge and develop a comprehensive solution, the Code Quality Fixer (CQF), an artificial-intelligence code review and improvement system, was used as a derivative of the particular research subject. An automated system is necessary for the comprehensive review and improvement of code quality using a large model.

In addition, the system allows users to input their corresponding source codes, and then the quality score may be generated, that is, the quality level or program reliability may be elaborated or the quality standards may be generated. Furthermore, it should be noted that the system would be effective in reviewing the codes to enhance or improve the codes in optimized forms and quality standards with the application of AI.

The proposed system would make this system interactive to perform code analysis with the codes given to the system. Graphs were plotted for the quality standards. It seems to be a highly effective system for helping programmers enhance their skills through feedback mechanisms rather than the traditional mechanism of code reviews and good codes for programmers. The proposed system can be extended by adding many features, such as different programming languages and security issues: Integration with Source Control Systems for Continuous Quality Monitoring.

Index Terms— Artificial Intelligence, Static Code Analysis, Automated Code Review, Code Quality Assessment, Software Engineering, Optimization, Developer Assistance, LLM, AST Parsing.

I INTRODUCTION

In relevance to the improvement of software application development in relation to the contemporary paradigm, that is, the software development paradigm, which has been adhered to in the development of software applications, it has been observed that with regard to the development of efficient and free from error codes, it has been regarded as one of the most important needs, that has been considered to be necessary to be accomplished, in relation to the development of software applications. Associated with this interpretation, it could also be added that with regard to the development of the levels of complexities related to software applications, there has been a need for maintaining issues related to the improvement of coding efficiency and developing codes of high quality, so that software developers, especially students, are put in a major way due to inadequate levels of expertise over the subject of programming along with optimum feedback mechanisms, forcing them to develop inefficient codes in relation to the development of software applications.

The quality of the code has generally improved through code review, which is usually carried out manually by skilled programmers/coders and by instructors. While effective, it is labour intensive, subjective, and often impossible, especially in larger groups and educational institutions with many

students, to implement. Code analysis tools have somewhat helped improve the standards of this activity, as they have been instrumental in automatically checking for syntax- or rule-based errors in the code. However, it is not possible for such tools to offer intelligent suggestions for improvement, as determined through programming logic and intelligent analyses.

To compensate for these gaps, this study aims to develop a Code Quality Fixer by introducing a set of new code analysis techniques with the help of artificial intelligence for code review to provide suggestions to the developer along the lines of quantifiable code qualities.

II LITERATURE SURVEY

Software quality assurance is an important area of research in software engineering that specifically deals with particular techniques to ensure the quality of the code and software. If we talk about particular aspects and techniques to ensure quality in the software, it has been found that the major basis of assessment of software quality lies within manual peer review. This particular task is time-consuming, and at the same time, there are specific biases that need correction. Generally, static code analyzers, such as SonarQube, Pylint, and Check style, are applicable in any environment, whether academic or industrial. This particular software is actually helpful for analyzing and assessing the complexities of the code written and, at the same time, can detect specific bugs that are included in developing that particular software. It has also been recognized that this particular software is intended to perform its functions without executing a source code. The tool does not understand any code behavior, nor does it permit intelligent suggestions that could enhance its performance. There have been a number of developments in recent times with respect to Artificial Intelligence and Natural Language Processing in creating intelligent machines that can understand programming logic and can propose a particular technique to optimize it. GPT, a type of Large Language Model, has been found to perform well in code completion, bug fixing, and documentation generation.

III SYSTEM ARCHITECTURE

A modular and scalable approach was followed to develop the architecture of the Code Quality Fixer.

1. Presentation Layer: This layer is responsible for generating the user interface using the Streamlit library. This layer is important because it enables a user to input and analyze the code
2. Application processing layer: Its functions include controlling and directing the workflow and code.
3. Static Code Analysis Module: It executable and employs the Abstract Syntax Tree parsing technique
4. Quality Scoring Engine: It used for health score calculations, considering maintainability and best practices
5. AI Review Module: As one can see, the plugin utilizes its association with a Large Language Model (LLM) while doing intelligent code review.

1. Data Management Layer: Here data are processed in real time without storage
2. Configuration and Security Layer: This will include configurations such as API configuration, among others. Workflow Summary

Automated AI Code Review and Analysis Pipeline: User Input → Static Analysis → Quality Score Calculation → AI Code Review → Result Aggregation → Visualization Output.

IV DATA FLOW DIAGRAM (DFD)

Figure 1 presents a general view of the process of data movement from the user input to the final intelligent feedback dashboard.

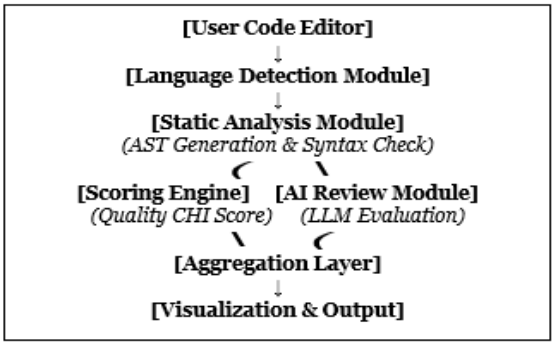


Figure 1: Graphical representation of system data flow.

1. Deep Explanation of Data Workflow

Based on the figure shown in the Data Flow Diagram above, it is evident that the figure provides a clear representation of the entire process through which the raw code inputs undergo transformation to produce useful feedback. Based on the figure, it is clear that the process starts by receiving the input through the User Code Editor. When the input is detected by the Language Detection Module, analysis is performed using the Static Analysis Module through non-executable analysis, generating an AST.

The process then takes a different course and branches off into two, with the role of the Scoring Engine being to create a score pertaining to quality. Simultaneously, with the use of the AI Review Module, the LLM will be applied to examine the code using optimization techniques. This information is then represented using the Visualization Module.

V METHODOLOGY AND FORMULATION

The approach provides an automated pipeline for reviewing source code quality, centered on static analysis and AI-based reasoning. Quality Score Formulation

The general scoring formula used in the system is expressed as follows:

$$QualityScore = \Sigma(Metric_i \times Weight_i \quad (1)$$

V.1.1 Meaning of the Formula

The final score is calculated by multiplying each individual quality metric by its assigned importance weight and summing the total.

V.1.2 Components of the Formula

- $Metric_i$: Individual code quality parameter.
- $Weight_i$: Relative importance assigned to the parameter.
- Σ: The total sum of all calculated weighted metrics.

VI IMPLEMENTATION DETAILS

Code Quality was achieved in a modular application, realizing separation between UI interactions, backend processing, and using AI. Streamlit was used as a library for front-end creation of user interfaces. This means that users can directly input the source code and visualize the results of the analysis. It achieves backend logic using Python modules to evaluate the abstract syntax tree (AST) for syntax analysis. Special libraries can also be used to analyze the code quality in terms of complexity. Conversely, AI can also be applied using LLM frameworks such as the Lang Chain. Communication occurs via an encrypted channel.

VII RESULTS AND EVALUATION

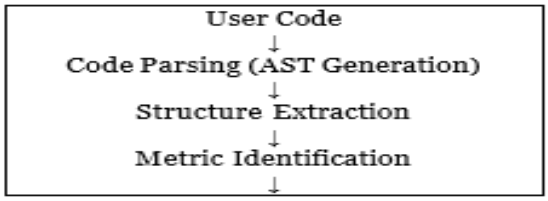
The performance was evaluated based on processing efficiency and feedback accuracy.

1. Semantic Analysis Performance

Semantic analysis interprets the code structure beyond the simple keywords

Table 1: Code Structure Identification Accuracy

Method	Accuracy (%)
Manual Checking	72%
Rule-Based Parsing	80%
Code Quality Fixer Analysis	90%

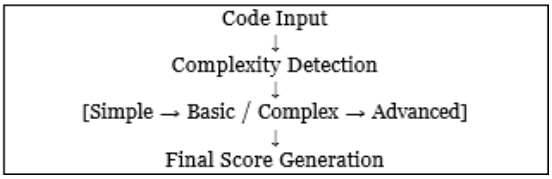


2. Adaptive Assessment Evaluation

The evaluation intensity was adapted based on the detected complexity levels.

Table 2: Adaptive Evaluation Metrics

Metric	Observed Value
Analysis Time	3–5
seconds Complexity Accuracy	88%
Maint. Accuracy	85%

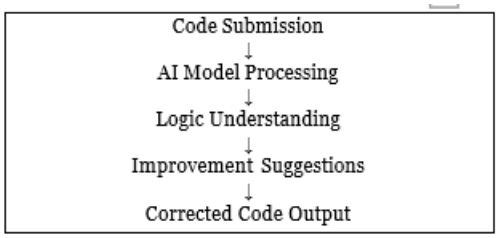


3. AI-Based Review Simulation

The AI module simulates expert reviews by identifying inefficient patterns.

Table 3: AI Review Performance Metrics

Parameter	Result
Error Detection	86%
Opt. Suggestions	84%
Acceptance Rate	82%



VIII CONCLUSION

It should be noted that the "Code Quality Fixer" project effectively demonstrates the capability for improvement in good coding practices.

static code analysis techniques by employing AI code reviews. With the increasing complexity of completing the software development process daily, the need to ensure proper code quality is also increasing. This, in turn, significantly affects the code in terms of its performance and efficiency. Although employing manual code review techniques might be helpful when used in the proposed system, it would take longer owing to the impractical nature of the entire process in educational institutions because of the complexity faced in the process itself. Thus, an efficient method for determining code quality is proposed.

It accepts user code submissions and analyzes the code in a structured manner without executing the code in any program state. Hence, it processes the code safely while detecting syntax, structural complexity, and maintainability-related issues in the code. In addition, the quality scoring feature provides an idea of the condition of the code developed by developers and students.

Most importantly, the integration of AI-related modules will allow intelligent suggestions to be provided to programmers in future improvements of the code and will even suggest optimized versions of the code. The evaluation results show that the code effectively detects issues in code quality and allows the user to correct inefficient and incorrect code patterns in programming. For this purpose, it may help improve coding skills using better suggestions and improvements than before. It would also allow the system to be flexible to extend in the future, allowing it to become an academic and professional development system. In conclusion, Code Quality Fixer helps fill the gap between manual code reviews and automated code evaluations by creating a realistic and easy-to-use tool/platform to improve programming quality. This project helps improve software development efficiency and simultaneously assists learners who are also programmers. This project contributes to high efficiency in software development and simultaneously assists learners who are also programmers.

IX FUTURE WORK

Although the main goal of the Code Quality Fixer tool is to perform code quality analysis in an automated way, there are opportunities to improve this tool further. The way to improve this tool further is considered to be an option because it supports different programming languages, such as C++, JavaScript, and Go, and it can be further improved in the future to support more programming languages. In other words, it supports different types of programming environments.

Other features that can be added in the future include readability and code smell.

The detection mechanisms could be used to perform some in-depth analysis on the code as well. In addition to this, the inefficient code duplication, inefficient loops, and design patterns in code could also be used to enhance the overall quality of the evaluation process.

Except for the said feature, it has also been identified that there exist some possibilities for incorporating vulnerabilities in terms of evaluation of the injection vulnerabilities and the resources. This again demonstrates that the identified scope for

the tool is applicable for secure code as well.
Moreover, it would also offer a facility to expand it for adding persistent storage as well as a user, wherein the programmer would have a chance to see how his improvement is over time, along with a report of the quality of code he is writing. Another way through which this system could improve is in relation to the improvement of the reasoning ability of the AI, in order to provide more specific optimization suggestions as well as reduce the inaccuracies involved in the process itself. Also, real-time collaboration would be added to the system, which could be very helpful to an education center like schools. Moreover, the system itself would be a Software as a Service.

ACKNOWLEDGEMENT

would like to extend our heartfelt thanks to our project guide and the faculty members for their continuous support and suggestions in the development of the Code Quality Fixer Project. This research work would not have been possible without their vested interest and contributions in a deep and impactful manner. There are moments in the development of the project, where the project would not have been developed without the "enhance and proper suggestions." We extend our heartfelt thanks to the Department of Computer Science and Engineering for providing us with the necessary infrastructural facilities and academic environment to develop the project successfully. The support and assistance we received from our fellow course mates, who too have been through the intense testing period, are invaluable. Lastly, we would like to place on record our appreciations to all the authors, developers, and contributors for their assistance in the successful implementation of the static code analyzer and the inclusion of the artificial intelligence concept in the project.

REFERENCES

[1] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed. Microsoft Press, 2004.
[2] M. Fowler *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.
[3] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.
[4] P. Oman and J. Hagemeister, "Metrics for Assessing Software System Maintainability," *Proceedings of the IEEE Conference on Software Maintenance*, 1992.
[5] Sonar Source, "SonarQube: Continuous Code Quality Inspection," Available: https://www.sonarqube.org.
[6] Pylint Development Team, "Pylint Documentation — Python Static Code Analysis Tool," Available: https://pylint.org
[7] Radon Developers, "Radon — Python Code Metrics Tool," Available: https://radon.readthedocs.io
[8] T. Brown et al., "Language Models are Few-Shot Learners," *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
[9] J. Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of NAACL-HLT*, 2019.
[10] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson Education, 2021.
[11] Lang Chain Documentation, "Building Applications with Large Language Models,"
[12] ISO/IEC *25010:2011, Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE).*