WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

AUTOMATED CODE INSPECTOR

SURYA R[1], SURYADHARSHINI S[2], SUSILA M [3], SWATHI M [4]

UG Students, Department of Computer Science and Engineering, Sri Shakthi Institute Of Engineering And Technology,Coimbatore - 641062.

ABSTRACT:

Software quality plays a crucial role in developing reliable, secure, and maintainable applications. Traditional code inspection methods require considerable time and manual effort to identify programming errors, coding standard violations, and potential security issues. To address these challenges, this project presents an Automated Code Inspector, an intelligent code analysis system designed to examine source code and provide detailed feedback. The system allows users to submit source code through a user-friendly interface for automated inspection. It analyzes the submitted code to identify syntax errors, coding style violations, code complexity, unused variables, and potential code quality issues. The backend processes the source code using analysis modules and AI-assisted evaluation to generate meaningful explanations and improvement suggestions. It also evaluates code quality based on identified issues and presents the results in a structured format. The system helps developers understand programming mistakes, improve code readability, reduce debugging effort, and maintain better coding practices. By combining automated analysis with intelligent feedback, the Automated Code Inspector provides an efficient and accessible approach to improving software quality and supporting the software development process.

Keywords: Code Analysis, Error Detection, Code Quality, AI Code Review, Code Optimization, Software Testing, Bug Detection.

INTRODUCTION

Software applications have become an essential part of everyday life, supporting various activities such as communication, education, banking, healthcare, and business operations. As software systems continue to grow, maintaining code quality, reliability, and security has become increasingly important. Writing code is not only about making a program work but also about ensuring that it is readable, efficient, maintainable, and free from common errors.

Code inspection is an important process in software development that helps identify mistakes and improve the overall quality of source code. Traditionally, developers depend on manual code reviews and debugging techniques to find errors. However, manually examining large amounts of code can be time-consuming and may lead to missed errors, inconsistent results, and increased development effort.

These challenges create a need for automated tools that can examine source code and provide useful feedback.

Automated code inspection helps overcome these difficulties by analyzing source code and identifying potential problems without requiring developers to check every line manually. It can detect syntax errors, coding style violations, unnecessary code, complexity issues, and other common programming mistakes. Such tools help developers understand problems earlier and improve their coding practices.

The Automated Code Inspector is developed to provide a simple and efficient platform for analyzing source code. Users can submit their code through a user-friendly interface, where the system examines it using different analysis techniques and AI-assisted evaluation. It identifies possible errors, evaluates code quality, and provides explanations and suggestions to improve the submitted program. The system also presents analysis results in a structured and understandable format.

The main aim of this project is to reduce manual code inspection effort and make code analysis easier for students and developers. By combining automated code analysis with intelligent feedback, the proposed system supports better programming practices, improves code readability, and helps produce more reliable and maintainable software.

WHAT IS CODE INSPECTION?

Code inspection is a systematic examination of source code intended to identify programming mistakes, coding standard violations, and potential weaknesses before software is tested or released. It can be performed manually through peer reviews and walkthroughs or automatically using static analysis tools.

Unlike software testing, code inspection does not necessarily require the program to be executed. It examines the source code structure and identifies possible problems based on predefined rules and analysis techniques. This makes it useful for detecting issues that may not be covered by normal testing.

Static code analysis is an automated form of code inspection. The analyzer reads the source code, creates an internal representation such as a token stream or syntax tree, and applies different rules and metrics to identify potential issues. Since the analysis follows predefined rules, it provides consistent and repeatable results.

Code inspection helps improve code readability, maintainability, efficiency, and security. It also helps developers understand programming mistakes and maintain coding standards throughout the development process.

NEED FOR AUTOMATED CODE INSPECTION

As software projects grow, manually examining every line of source code becomes increasingly difficult. Developers may work with multiple files, functions, classes, and programming structures, making the inspection process time-consuming.

Automated code inspection helps address these challenges by providing faster and more consistent feedback. It applies the same inspection rules across different programs and reduces the effort required for routine code reviews.

The need for automated code inspection can be explained through the following points:

- Time Efficiency: Reduces the time required to identify common programming issues.
- Consistent Analysis: Applies predefined inspection rules uniformly across source code.
- Error Identification: Helps detect syntax errors, coding violations, and unnecessary code.
- Code Quality: Identifies complexity and code structure problems that may affect maintainability.
- Security Awareness: Highlights common unsafe coding practices and potential vulnerabilities.
- Learning Support: Helps students understand programming mistakes and improve their coding practices.

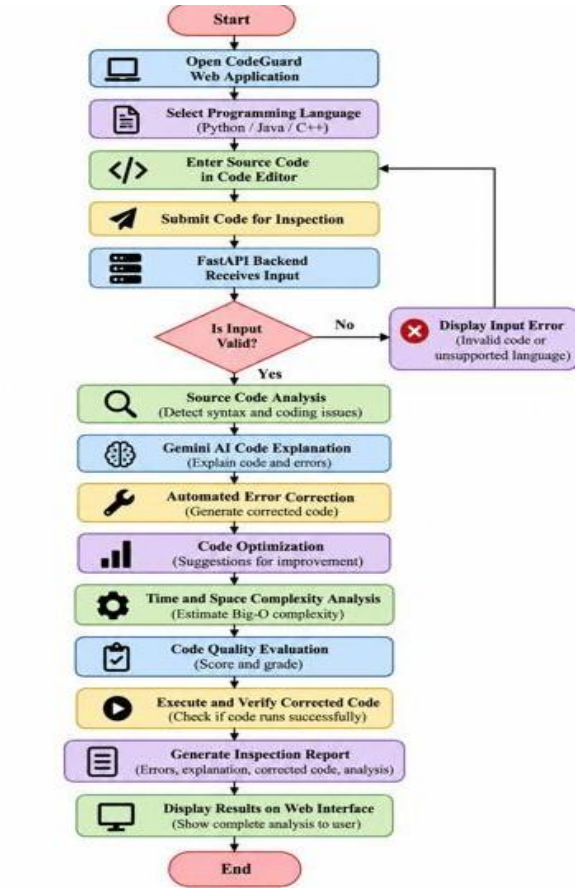
Automated inspection does not replace human code reviews completely. Instead, it handles routine checks and allows developers to concentrate on program logic, design, and application requirements.

METHODOLOGY

The Automated Code Inspector follows a modular pipeline architecture. The submitted source code is first processed and converted into an Abstract Syntax Tree. The resulting structure is examined by multiple independent analysis modules. Each module identifies a particular category of issue and sends its findings to a central issue aggregator.

The aggregator combines the findings, removes duplicate reports, assigns severity levels, and calculates an overall code quality score. Finally, the report generation module presents the analysis results in a structured and understandable format.

The modular architecture allows individual analysis components to be modified or extended without affecting the entire system. The current prototype focuses on Python source code.



A. Source Code Input and Parsing

The source code input module accepts the program submitted by the user for inspection. The input is processed before analysis to maintain a consistent representation of the source code.

The parser converts the source code into an Abstract Syntax Tree (AST), which represents program elements such as functions, classes, loops, conditions, and expressions in a hierarchical structure.

The AST provides a structured representation that allows the analysis modules to examine the program more effectively than checking raw text alone. If the submitted program contains a syntax error, the system reports the error along with its corresponding line number.

B. Style Checker

The style checker examines the source code for violations of commonly accepted Python coding conventions. It identifies issues that may reduce readability and make program maintenance difficult.

The module checks for inconsistent indentation, trailing whitespace, lengthy lines, improper naming conventions, wildcard imports, and missing documentation strings.

By identifying these issues, the style checker encourages consistent coding practices and improves the overall readability of the source code.

C. Complexity Analyzer

The complexity analyzer examines the structural complexity of functions and identifies code that may be difficult to understand, test, and maintain.

It uses cyclomatic complexity to estimate the number of independent execution paths in a program. The metric is calculated using the formula:

$V(G) = E - N + 2P$

Where:

E represents the number of edges in the control flow graph. N represents the number of nodes in the control flow graph. P represents the number of connected components.

The analyzer also examines function length, nesting depth, and other structural measurements. Functions with high complexity values may require additional attention or restructuring.

D. Report Generation

The report generation module presents the inspection results in a structured format that can be easily understood by users.

The report includes the overall code quality score, the number of detected issues in each category, severity levels, affected line numbers, issue descriptions, and suggested improvements.

The system is designed to provide analysis results through readable reports and structured output formats. This helps developers understand the identified problems and take appropriate corrective actions.

KEY FEATURES

The Automated Code Inspector provides the following key features:

1. Accepts Python source code as input for automated inspection.
2. Identifies syntax errors and coding style violations.
3. Examines cyclomatic complexity and code nesting.
4. Detects potential code smells such as long functions and duplicated logic.
5. Identifies unused and unreachable code.
6. Scans for common security-related coding weaknesses.
7. Classifies identified issues according to severity.
8. Calculates an overall code quality score.
9. Generates structured reports with issue descriptions and suggested improvements.
10. Uses a modular architecture that supports the extension of analysis modules.

OBJECTIVE

The primary objective of this project is to develop an Automated Code Inspector that analyzes source code and identifies potential programming issues without requiring program execution.

The system aims to detect coding style violations, unused code, code smells, and common programming errors through systematic source code analysis.

Another objective is to evaluate the complexity of the source code and identify sections that may affect program readability, maintainability, and overall software quality.

The project also focuses on identifying potential security vulnerabilities through rule-based scanning, helping developers recognize unsafe coding practices and improve code security.

Furthermore, the system aims to classify the detected issues based on their severity and generate a structured report containing issue descriptions, affected line numbers, and suggested improvements.

It also aims to calculate an overall code quality score based on the identified issues, enabling developers to assess the quality of their source code and make necessary improvements.

Finally, the project intends to reduce the time and effort involved in manual code inspection while promoting better coding practices and the development of reliable, maintainable software.

RESULTS

The Automated Code Inspector was evaluated using sample Python programs containing deliberately introduced coding issues. The original project document reports testing across five categories: style and naming violations, excessive complexity, unused or unreachable code, code smells, and security weaknesses.the reported evaluation considered the number of seeded issues, detected issues, false-positive warnings, and detection rates for each category.

Table 1. Reported Detection Results of the Automated Code Inspector

Check Category	Seeded Issues	Detected Issues	False Positives	Detection Rate
Style and Naming	40	40	2	100%
Complexity	25	24	1	96%
Unused / Dead Code	20	19	1	95%
Code Smells	30	27	3	90%
Security	35	31	2	88.6%
Overall	**150**	**141**	**9**	**94%**

According to the reported results, the system detected 141 out of 150 seeded issues, achieving an overall detection rate of 94%.

Style and naming violations showed the highest detection rate because these checks depend mainly on identifiable coding patterns. Complexity and unused-code analysis also produced strong results because they use structural information from the source code.the code smell detector produced comparatively more false-positive warnings because certain structural conditions, such as function length and nesting depth, may depend on project requirements and developer preferences.

The security scanner showed the lowest detection rate among the five categories. This limitation is associated with the use of rule-based pattern matching, which may not identify vulnerabilities that require tracking data across multiple assignments or function calls.

Overall, the reported evaluation demonstrates the potential of automated static analysis to identify common source code issues and provide consistent inspection results. However, automated inspection cannot fully replace human review, particularly when evaluating application requirements, design decisions, and complex program logic.

CONCLUSION

The Automated Code Inspector provides a systematic approach to examining source code and identifying potential programming issues through static analysis. The system uses an Abstract Syntax Tree to represent the submitted program and applies independent analysis modules for style checking, complexity analysis, code smell detection, unused-code identification, and security scanning.

The issue aggregator combines the findings, classifies them according to severity, and calculates an overall code quality score. The report generation module presents the results with issue descriptions, affected line numbers, and suggested improvements.

The proposed system aims to reduce the effort required for routine manual code inspection and provide students and developers with consistent feedback about their source code. Although static analysis can identify several common programming issues, it has limitations in detecting complex security vulnerabilities and understanding application-specific logic. Therefore, combining automated inspection with human code review can provide a more effective approach to improving software quality.

REFERENCES

1. Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 182–211.
2. Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. *Proceedings of the 35th International Conference on Software Engineering*, 712–721.
3. Johnson, S. C. (1978). Lint, a C program checker. *Bell Laboratories, Computing Science Technical Report 65*.
4. Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J., & Pugh, W. (2008). Using static analysis to find bugs. *IEEE Software*, 25(5), 22–29.
5. Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? *Proceedings of the 35th International Conference on Software Engineering*, 672–681.
6. Chess, B., & McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy*, 2(6), 76–79.
7. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.
8. Python Software Foundation. *ast – Abstract Syntax Trees*. Python Standard Library Documentation.
9. van Rossum, G., Warsaw, B., & Coghlan, N. (2001). PEP 8 – Style Guide for Python Code. Python Enhancement Proposals.

10. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308–320.
11. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6), 476–493.
12. Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
13. OWASP Foundation. *OWASP Top Ten Web Application Security Risks*. OWASP.
14. Li, Z., et al. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. *Proceedings of the Network and Distributed System Security Symposium*.