

Large Language Models for Automated Software Code Generation, Optimization, and Bug Detection

Ashwini. H

Abstract

Large Language Models (LLMs) have emerged as an important advancement in Artificial Intelligence, particularly in the field of automated software engineering. Their ability to understand natural-language instructions and generate programming code has created new opportunities for automating software development activities. This paper investigates the application of LLMs in three important areas of software engineering: automated code generation, code optimization, and bug detection. The proposed approach establishes an integrated LLM-based workflow in which software requirements expressed in natural language are transformed into source code, followed by automated validation, optimization, and defect identification. The methodology combines prompt-based generation with static analysis and automated testing to improve the reliability of generated programs. The evaluation framework considers functional correctness, code quality, execution efficiency, bug-detection capability, and testing effectiveness. The study also examines the limitations of LLM-based programming, including hallucinated code, logical inconsistencies, security vulnerabilities, and incorrect interpretation of requirements. Rather than replacing software developers, LLMs are considered intelligent programming assistants that can reduce repetitive development effort and support developers during different stages of the software development lifecycle. The proposed framework provides a systematic foundation for evaluating LLM-assisted programming and highlights the importance of combining generative AI with automated validation and human supervision.

Keywords: Large Language Models, Artificial Intelligence, Code Generation, Code Optimization, Bug Detection, Generative AI, Software Engineering, Automated Programming.

Received : 10.06.2026

Acceptance : 15.06.2026

Publication : 17.06.2026

1. INTRODUCTION

Software development has traditionally been a human-intensive process involving requirement analysis, program design, coding, testing, debugging, optimization, and maintenance. As modern software applications have become more complex, development teams increasingly face challenges related to development time, programming effort, software quality, security, and maintenance costs. The increasing demand for software applications has therefore created a need for intelligent technologies capable of assisting developers throughout the software development lifecycle. Artificial Intelligence has become an important technology for addressing these challenges, particularly through machine learning, deep learning, natural language processing, and generative AI.

Large Language Models represent a major development in generative artificial intelligence. These models are trained on large-scale textual and programming datasets and can understand patterns in both natural language and source code. Unlike traditional programming assistants that rely primarily on predefined rules or templates, LLMs can interpret natural-language descriptions and produce programming solutions.

A developer can describe a required functionality using ordinary language, specify the programming language and constraints, and obtain a candidate implementation from an LLM. This capability has significantly increased interest in AI-assisted software development.

One of the most important applications of LLMs is automated code generation. LLMs can generate functions, classes, algorithms, database queries, scripts, and complete program components from natural-language descriptions. They can also complete partially written code and convert programs from one programming language to another. Such capabilities can reduce the amount of repetitive coding required from developers. However, generated code is not guaranteed to be correct. An LLM can produce syntactically valid code that contains logical errors, inappropriate assumptions, inefficient algorithms, or security weaknesses. Consequently, code generation should be considered a candidate-generation process rather than a complete replacement for software development.

LLMs can also support code optimization and bug detection. During optimization, an LLM can examine existing source code and suggest alternative algorithms, simplified expressions, improved data structures, or more efficient programming practices. During debugging, the model can analyze source code and error messages, identify possible causes of defects, and recommend corrections. The combination of these capabilities creates the possibility of an integrated AI-assisted development environment. However, the effectiveness of such systems depends on the quality of prompts, model capabilities, programming language, complexity of the task, and availability of automated validation mechanisms.

This study therefore proposes an integrated framework for using LLMs for automated code generation, optimization, and bug detection. The objective is not to eliminate human developers but to investigate how generative AI can support developers by automating repetitive and time-consuming programming activities. The proposed framework combines LLM generation with static analysis and automated testing to improve reliability and provide a systematic evaluation process.

2. RELATED WORK

The application of Artificial Intelligence to software engineering has been studied for several decades, initially through rule-based programming assistants, automated program synthesis, static analysis, and machine-learning-based defect prediction. Traditional automated programming approaches generally depend on predefined rules, templates, formal specifications, or specialized algorithms. Although these approaches can perform specific tasks effectively, they often have limited flexibility when requirements are expressed using natural language. The emergence of transformer-based language models has created a fundamentally different approach in which programming knowledge can be learned from large collections of source code and natural-language documentation.

The development of transformer architectures significantly influenced natural-language processing and subsequently programming-related applications. Transformer-based models are capable of learning contextual relationships within sequences and can therefore process programming languages in addition to natural languages. Code-oriented language models demonstrated that source code can be represented and generated using techniques similar to those used for natural-language generation. This development created the foundation for modern AI programming assistants.

Research on neural code generation has shown that machine-learning models can generate program fragments based on natural-language descriptions or incomplete code. Models trained specifically on programming repositories have demonstrated the ability to predict functions, complete code, and generate solutions to programming problems. Such systems can reduce developer effort, particularly for routine programming tasks. Nevertheless, several studies have emphasized that functional correctness

remains a major challenge because generated programs may appear plausible while failing particular test cases.

LLMs have also been investigated for software testing and debugging. Their natural-language reasoning capabilities allow them to analyze error messages, explain program behavior, generate test cases, and propose bug fixes. In particular, LLMs can be useful when developers need explanations rather than only error indicators. A traditional static-analysis tool may identify a suspicious statement, whereas an LLM can potentially explain why the statement is problematic and provide an alternative implementation. However, LLM-generated explanations can themselves be inaccurate and therefore require validation.

Another important research direction is automated code optimization and refactoring. Software maintenance frequently requires developers to restructure existing programs without changing their intended functionality. LLMs can propose refactoring operations, improve code readability, remove duplicated logic, and suggest alternative implementations. The challenge is ensuring that optimization or refactoring does not alter the intended behavior of the original program. Automated testing is therefore essential for verifying semantic equivalence after an AI-generated modification.

Existing research indicates that LLMs have considerable potential in software engineering, but reliability remains an important research problem. The current study addresses this issue by combining code generation, optimization, bug detection, static analysis, and automated testing into a single workflow. Such integration provides a more practical approach than evaluating code generation independently because real-world software development involves continuous interaction between these activities.

3. PROPOSED METHODOLOGY

The proposed methodology is designed as an integrated LLM-based software engineering framework. The framework accepts a software requirement in natural language and progressively transforms it into validated source code. Instead of directly accepting the output produced by the LLM, the proposed approach introduces validation and refinement stages. This allows syntactic errors, functional defects, and potential optimization problems to be identified before the final code is presented to the developer.

The first stage involves requirement processing. The user provides a natural-language description of the desired software functionality. The requirement is analyzed to identify the programming language, expected inputs and outputs, functional constraints, performance requirements, and error-handling conditions. These requirements are then incorporated into a structured prompt. Effective prompt construction is important because incomplete or ambiguous instructions can result in inappropriate code generation.

The second stage is LLM-based code generation. The structured prompt is provided to the selected LLM, which generates source code according to the specified requirements. Depending on the application, the generated output may include a complete program, a function, a class, a database query, or a specific algorithm. The generated code is then passed to a validation layer.

The validation layer performs syntax checking and static analysis. Syntax checking determines whether the generated program can be compiled or interpreted successfully. Static analysis examines the source code for potential defects, insecure programming patterns, unused variables, unreachable code, resource-management problems, and other quality issues. If significant problems are detected, the code is returned to the refinement stage.

After static validation, automated testing is performed. Test cases are executed against the generated implementation to determine whether it satisfies the expected functional behavior. Test cases should include normal inputs, boundary conditions, invalid inputs, and potentially unexpected scenarios. If the

implementation fails one or more tests, the failure information can be provided back to the LLM for code refinement.

The optimization stage examines the validated implementation for possible performance and maintainability improvements. The LLM can suggest algorithmic modifications, reduction of redundant operations, improved data structures, or simplified implementation strategies. Each optimization must be evaluated using the same functional test suite to ensure that the modified implementation continues to produce correct results.

Finally, the system performs bug detection and quality assessment. Potential defects identified by the LLM are compared against static-analysis results and test outcomes. The final code is presented to the developer together with explanations of detected problems and recommended improvements. Human validation remains the final decision-making stage because software requirements can involve contextual information that may not be fully available to the model.

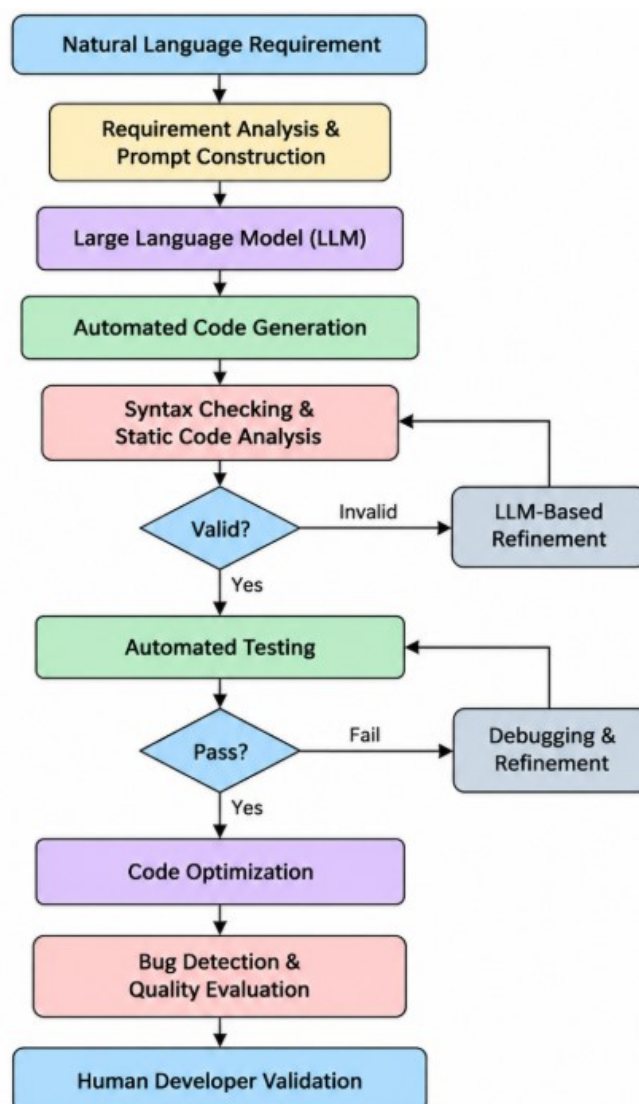


Figure 1. Proposed LLM-Based Software Engineering Framework

4. LLM-BASED CODE GENERATION, OPTIMIZATION AND BUG DETECTION

4.1 Automated Code Generation

Automated code generation is the primary capability of the proposed framework. The process begins with a natural-language requirement describing the desired functionality. The LLM interprets the requirement and generates a corresponding programming implementation. The generated code can contain functions, classes, algorithms, input-processing mechanisms, and error-handling logic depending on the complexity of the requirement.

The effectiveness of code generation depends on both model capability and prompt quality. A detailed prompt containing explicit requirements generally provides a more reliable basis for generation than a highly ambiguous instruction. Context can also influence the generated implementation. For example, specifying the programming language, expected input format, output format, computational constraints, and security requirements can reduce ambiguity.

Despite its advantages, automated code generation presents a significant reliability challenge. A generated program may compile successfully but still fail under particular input conditions. Therefore, functional correctness should be established through automated testing rather than by examining the generated source code alone. The proposed framework addresses this issue by making automated testing an integral component of the generation process.

4.2 Code Optimization

Code optimization focuses on improving the efficiency and quality of software while preserving its intended functionality. LLMs can analyze existing programs and recommend improvements based on algorithmic structure and coding patterns. For example, a model may identify repetitive calculations, inefficient loops, unnecessary data processing, or inappropriate data structures.

Optimization should be evaluated using measurable performance indicators. Execution time can be measured before and after optimization, while memory consumption can be monitored during program execution. Code complexity and maintainability can also be considered. An optimization is considered useful only when it improves one or more relevant characteristics without introducing functional errors.

The proposed framework therefore applies the same test suite to both original and optimized versions. This ensures that performance improvements do not come at the expense of correctness. The approach also allows developers to compare multiple optimization alternatives before selecting the final implementation.

4.3 Automated Bug Detection

Bug detection is performed using a combination of LLM analysis, static analysis, and automated testing. The LLM examines source code and identifies possible logical, runtime, security, and maintainability problems. Static-analysis tools provide an additional layer of verification by detecting patterns associated with known programming defects.

Automated testing provides behavioral evidence regarding whether the program performs according to the expected requirements. When a test fails, the failure information can be supplied to the LLM, allowing the model to propose a correction. The corrected program is then tested again.

This iterative process can be repeated until the program satisfies the predefined validation criteria or reaches a specified iteration limit. The combination of AI-generated reasoning and automated verification is important because LLM predictions alone may contain false positives or false negatives.

5. EXPERIMENTAL SETUP AND EVALUATION

The proposed framework can be evaluated using a collection of programming tasks covering different levels of complexity. The experimental dataset may include algorithmic problems, data-processing tasks, object-oriented programming problems, database operations, and debugging tasks. Each task should have a predefined functional specification and a test suite that can be used as ground truth for evaluation.

A selected LLM is provided with standardized prompts for each programming task. The generated programs are executed in a controlled environment. Syntax correctness is determined through compilation or interpretation, while functional correctness is determined through automated test execution. For optimization experiments, the execution time and memory consumption of the original and optimized implementations are measured under identical conditions.

The evaluation should focus on several dimensions rather than a single accuracy measure. Functional correctness indicates whether generated code produces the expected results. Bug-detection performance indicates how effectively the system identifies actual defects. Optimization effectiveness measures changes in execution efficiency. Code quality can be evaluated through complexity, maintainability, and static-analysis warnings. Developer effort can additionally be estimated by measuring the amount of manual modification required before the generated code becomes acceptable.

Table 1. Proposed Evaluation Framework

Evaluation Dimension	Measurement	Purpose
Functional Correctness	Test-case pass rate	Measures whether generated code satisfies requirements
Syntax Correctness	Compilation/interpreter success	Identifies syntactic failures
Bug Detection	Precision, recall, F1-score	Measures defect-identification capability
Execution Efficiency	Execution time	Evaluates optimization
Memory Efficiency	Peak memory usage	Measures resource utilization
Code Quality	Static-analysis indicators	Evaluates maintainability and quality
Developer Effort	Manual correction required	Measures practical usefulness

For bug detection, precision and recall are used to evaluate the effectiveness of the proposed LLM-based approach. Precision measures the proportion of correctly identified bugs among all bugs reported by the system, while recall measures the proportion of actual bugs that are successfully detected. Precision and recall are calculated using the following equations:

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

where **TP (True Positive)** represents the number of correctly detected bugs, **FP (False Positive)** represents the number of incorrectly reported bugs, and **FN (False Negative)** represents the number of actual bugs that were not detected by the system.

The overall balance between precision and recall is measured using the F1-score, which is calculated as:

$$\text{F1-score} = 2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$$

A higher precision indicates that the system produces fewer false alarms, while a higher recall indicates that the system is capable of identifying a larger proportion of actual software defects. Therefore, these metrics provide a comprehensive evaluation of the LLM's effectiveness in automated bug detection

6. RESULTS AND DISCUSSION

The proposed evaluation framework is intended to determine how effectively LLMs can support different software engineering tasks. Because LLM performance depends on the selected model, programming language, prompt structure, benchmark dataset, and execution environment, numerical performance values should be obtained through actual experimentation rather than assumed in advance.

The expected evaluation is likely to show that LLMs provide substantial assistance in routine code-generation tasks where requirements are clearly defined. Problems involving standard algorithms and commonly used programming constructs are generally more suitable for automated generation than highly specialized applications requiring extensive domain knowledge. The effectiveness of the generated code should nevertheless be confirmed through automated testing.

For optimization, the comparison between baseline and LLM-optimized implementations can provide evidence regarding whether AI-generated modifications produce measurable performance improvements. An important observation is that optimization quality cannot be judged only from the appearance of the source code. Execution-time and memory measurements are required to establish whether a modification provides an actual improvement.

Bug detection should be analyzed using both detection metrics and qualitative examination of identified defects. LLMs can potentially identify problems that are difficult to express through simple static-analysis rules because they can interpret relationships between different sections of code. At the same time, they may report issues that are not actual defects. Combining LLM predictions with static-analysis tools and automated tests can therefore improve reliability.

The results should ultimately demonstrate whether integrating LLMs with automated validation provides greater reliability than using LLM-generated code without verification. The expected contribution of the framework is therefore not simply faster code generation, but a structured process in which generation, testing, debugging, and optimization operate together.

7. CHALLENGES AND LIMITATIONS

Despite the significant potential of LLMs in software engineering, several limitations need to be considered. The first challenge is the possibility of generating incorrect or hallucinated code. LLMs generate outputs based on learned patterns and probabilities rather than executing a formal proof of program correctness. Consequently, a generated implementation can appear highly plausible while failing under particular circumstances. This limitation makes automated testing and human review essential.

Security is another important concern. An LLM may generate code containing insecure programming practices, outdated libraries, improper input validation, or vulnerable authentication mechanisms. Developers therefore need to evaluate generated code using established security-analysis tools before deploying it in production environments. Privacy and intellectual-property considerations may also become relevant when source code or proprietary requirements are processed through external AI systems.

A further limitation concerns context and domain knowledge. Complex software projects may contain large codebases, organization-specific rules, legacy dependencies, and undocumented business logic. An LLM may not have sufficient context to understand all of these factors. In addition, optimization

recommendations can sometimes change program behavior or fail to provide measurable performance benefits. These limitations indicate that LLMs should function as developer-assistance systems rather than autonomous replacements for software engineers.

8. CONCLUSION

This research examined the application of Large Language Models to automated software code generation, code optimization, and bug detection. LLMs provide an important opportunity to transform conventional software development by allowing programmers to interact with programming systems through natural-language instructions. Their ability to generate, explain, modify, and analyze source code can reduce repetitive programming activities and provide developers with intelligent assistance throughout the software development lifecycle.

The proposed framework integrates LLM-based generation with syntax validation, static analysis, automated testing, optimization, and bug detection. This integrated approach is important because generated code cannot be assumed to be correct simply because it is syntactically valid or appears logically plausible. Automated testing and analysis provide objective mechanisms for validating AI-generated solutions, while human developers remain responsible for final verification and deployment decisions.

Future research can extend this framework by evaluating multiple LLM architectures, programming languages, benchmark datasets, and prompt-engineering strategies. Further work can also investigate secure code generation, vulnerability-aware optimization, repository-level code understanding, automated test generation, and agent-based software development. With appropriate validation and human supervision, LLMs have the potential to become an important component of intelligent and efficient software engineering environments.

REFERENCES

1. A. Vaswani et al., "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.
2. T. Brown et al., "Language Models are Few-Shot Learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.
3. M. Chen et al., "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.
4. D. Hendrycks et al., "Measuring Coding Challenge Competence With HumanEval," *arXiv preprint arXiv:2107.03374*, 2021.
5. M. Le et al., "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021.
6. Y. Wang et al., "CodeT5+: Open Code Large Language Models for Code Understanding and Generation," *arXiv preprint arXiv:2305.07922*, 2023.
7. D. Fried et al., "InCoder: A Generative Model for Code Infilling and Synthesis," *International Conference on Learning Representations*, 2023.
8. Vishram Singh, 2026. "Enterprise Resource Planning for Financial Applications: Engineering, Architecture, and Design Principles," *Int J Sci Res Sci Technol*, International Journal of Scientific Research in Science and Technology, vol. 13(2), pages 653-661, April.

9. N. Shinn et al., "Reflexion: Language Agents with Verbal Reinforcement Learning," *Advances in Neural Information Processing Systems*, 2023.
10. S. P. P. et al., "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Computing Surveys*, 2024.
11. M. Fan et al., "Large Language Models for Code Generation: A Survey," *ACM Transactions on Software Engineering and Methodology*, 2024.
12. B. Xia et al., "Automated Program Repair with Large Language Models," *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2024.
13. J. Li et al., "StarCoder: May the Source Be With You!" *Transactions on Machine Learning Research*, 2023.
14. R. Nijkamp et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," *International Conference on Learning Representations*, 2023.
15. Singh, V. (2026). Real-Time Financial Data Validation Platform for Enterprise Accounting Systems. *International Journal of Computer Information Systems and Industrial Management Applications*, 18(10s), 467-481.
16. Vishram Singh. (2026). Prompt Context Caching Architecture for Cost Reduction in Large Language Model Systems. *International Journal of Intelligent Systems and Applications in Engineering*, 14(1s), 1543 –. Retrieved from <https://www.ijisae.org/index.php/IJISAE/article/view/8385>
17. Todupunuri, A. (2025). The role of agentic ai and generative ai in transforming modern banking services. *American Journal of AI Cyber Computing Management*, 5(3), 85-93.
18. Todupunuri, A. (2025). The role of human-centric AI in building trust in digital banking ecosystems. *International Journal of Innovative Science and Research Technology*, 10(1), 1281-1286.
19. Bajarang Bhagwat, V. (2023). *Optimizing Payroll to General Ledger Reconciliation: Identifying Discrepancies and Enhancing Financial Accuracy*. *JOURNAL OF ADVANCE AND FUTURE RESEARCH*, 1 (4).
20. Bhagwat, V. B. (2025). Simplifying Payroll Balance Conversions in Payroll Systems Implementation through the Use of Generative AI.