A Review of Research on AI-Assisted Code Generation and AI-Driven Code Review

Yuzhi Wang

Beijing University of Technology, Beijing, China

shenrenaisite@yeah.net

Abstract. With the significant breakthroughs of deep learning technologies such as large language models (LLMs) in the field of code analysis, AI has evolved from an auxiliary tool to a key technology that deeply participates in code optimization and resolving performance issues. As modern software system architectures become increasingly complex, the requirements for their performance have also become more stringent. During the coding stage, developers find it difficult to effectively identify and resolve potential performance issues using traditional methods. This review focuses on the application of artificial intelligence in two key areas: AI-assisted intelligent code generation and AI-povered code review. The review systematically analyzed the application of LLMs in software development, revealing a situation where efficiency gains coexist with quality challenges. In terms of code generation, models such as Code Llama and Copilot have significantly accelerated the development process. In the field of code review, AI can effectively handle code standards and low-severity defects. However, in the future, this field still needs to address the issues of the reliability and security of the code generated by LLMs, as well as the insufficient explainability of the results of automated performance analysis. The future research focus in this field lies in addressing issues such as the lack of interpretability and insufficient domain knowledge of LLMs. It is necessary to prioritize enhancing the reliability of AI recommendations and promoting the transformation of AI from an auxiliary tool to an intelligent Agent with self-repair capabilities, in order to achieve a truly efficient and secure human-machine collaboration paradigm. This article systematically reviews the relevant progress, aiming to promote the transformation of software engineering from an artificial-driven model to an AI-enhanced automated paradigm. It provides theoretical references for ensuring the quality of backend code, improving product delivery speed, and enhancing system reliability.

Keywords: AI; LLM; Code Generation; Code Review.

1. Introduction

Recently, the growing complexity of modern software applications has driven an increased emphasis on high-quality, maintainable source code, thereby heightening the difficulty for developers to write efficient and error-free code [1-3]. Therefore, there is an urgent need for a smarter approach to empover developers. The emergence of Artificial Intelligence has fundamentally transformed conventional approaches to code optimization and refactoring by introducing a new dimension of automation [2]. A striking example is LLMs, which have exhibited remarkable potential in programming tasks [4], especially in code review [5] and code generation [6]. This review aims to systematically review and analyze the current application status and research progress of AI in software program development and code generation and review. The analysis focuses on two aspects: one is AI-assisted intelligent code generation, and the other is AI-powered code review. Through in-depth exploration of these cutting-edge studies, the value of this review lies in clarifying how AI can assist developers in performing more convenient programming tasks and enabling the early detection of program defects as well as the control of the root causes of performance issues. This not only significantly enhances the efficiency and code delivery speed of the development team, but more importantly, it greatly improves the reliability and quality of the final software program [7], thereby reducing operational costs and driving the entire software engineering field towards a higher-level intelligent-assisted development paradigm.

2. Literature Review

In recent years, the field of software engineering has developed rapidly. Software engineering encompasses the systematic and controlled design, development, maintenance, implementation and evolution of software systems [1]. As artificial intelligence has become a highly popular field at present, it has also become a new component of software engineering [8]. To assist in software development, AI is now widely used in scenarios such as code generation and code review [9]. In particular, large language models, as the latest breakthrough in the field of natural language processing in AI, have made particularly significant contributions to software development. LLM mainly employs the Transformer model as its core architecture. The LLM, which is built on large datasets and advanced neural network architectures, demonstrates extremely high comprehension capabilities, bringing about significant breakthroughs and possibilities to the field of software development [10]. For instance, models such as Codex [11], StarCode [12], Incoder [13], and Code Llama [14], Copilot[15]can all generate code with efficiency and accuracy comparable to that of developers. These LLMs not only have extremely high efficiency but also excellent quality in code review [10].

2.1 AI-Assisted Intelligent Code Generation

Code generation is an automated process that converts structured or unstructured input information (such as natural language requirements descriptions, design documents, code snippets, etc.) into source code. Its essence is to reflect the abstract intentions and task goals of the developers into specific programming projects [16]. And based on LLM (Large Language Model) for code generation, by breaking down the tasks, having data storage with long-term and short-term memories, as well as the invocation of external tools, these are currently important technical supports in the field of code generation [16]. This section will summarize the application effects and code generation quality of Codex and Copilot in the field of code generation.

2.1.1 Code Llama

The Code Llama model series released by Meta AI focuses on code generation through long sequence contexts and instruction fine-tuning, and its mechanism is of great significance for the generation of complex logic at the back end [14].

2.1.1.1 Strong Context-Dependent and Long Sequence Processing Capabilities

Modern backend development typically involves complex distributed systems and microservice architectures, and code generation must take into account global dependencies and architectural specifications. Code Llama uses a sequence length of up to 16k tokens during training, and is capable of optimizing to support inputs of up to 100k tokens. This long-context support capability is crucial for backend development, as it enables the model to understand global dependencies and adhere to architectural conventions when generating new functions or modules [14].

2.1.1.2 Zero-Shot Instructions Follow Natural Language Programming

The Code Llama Instruct variant in the Code Llama series has been specifically optimized for human natural language programming instructions, featuring a powerful "zero-shot instruction following ability" [14]. This means that developers can directly describe complex business logic or functional requirements using informal natural language (for example: "Read the CSV file, only filter the 7B model data, and draw a correlation heatmap of the Python and C++ fields") [14]. The model can accurately convert these instructions into complete, executable functions or scripts, significantly enhancing the efficiency from requirements to implementation. This capability is particularly applicable to common tasks in backend development such as API route definitions, parameter validation logic, or data processing functions.

2.1.1.3 Code Filling Capability

Code Llama supports the code filling function based on the surrounding content, allowing developers to set the boundaries of the code first, and then let the model automatically fill in the missing business logic in the middle according to the context of the previous and following text [14]. This provides core support for developers to carry out incremental development or iterative optimization in existing code

2.1.2 Copilot

As the earliest and most widely used AI pair programming tool in the market, the core advantage of GitHub Copilot (based on the OpenAI Codex model) lies in its massive data training and real-time code prediction mechanism [17].

2.1.2.1 Real-Time Context-Aware Prediction

Its real-time context-aware prediction mechanism focuses on the immediacy based on the cursor position, predicting the code snippet that best matches the current input. It is particularly good at generating repetitive template code, data structure initialization, and common I/O operation procedures [15].

2.1.2.2 Translation of Comments into Code

One of the key features of Copilot is the ability to generate code based on natural language prompts or partial code input [17]. Developers usually provide natural language comments above functions or code blocks to describe the required functionality. Copilot can then generate the corresponding function signatures, parameter definitions, and main logical framework based on this description. Although its instruction-following ability may not be as strong as the specially fine-tuned Instruct model, it performs well in implementing the functions based on the comment prompts [17].

Table 1. Comparative Analysis of Code Llama and Copilot in Code Generation

Aspect	Code Llama	GitHub Copilot
Core Positioning	Open Foundation Model, focused on model performance and customization.	Commercial Integration Tool, focused on user experience and real-time development efficiency.
Context Capability	Supports inputs up to 100k tokens, highlighting strong long-sequence context ability	Relies on a longer context window, analyzing code and documentation in the IDE in real-time.
Instruction Following	Strong zero-shot instruction following ability, directly converting natural language requests into complex code	Triggered by comments or function signatures, generating code completion or suggestions.
Quality Focus	Enhancing SOTA performance and generalizability, providing open-source community support.	Improving developer productivity and usability, addressing low-severity code issues

By examining the mechanisms of Code Llama and GitHub Copilot, it can be seen that the application of LLM in the field of code generation has demonstrated characteristics of efficiency-driven and specialized division of labor.

2.2 AI-Powered Code Review

Code review is a crucial stage in the software development life cycle, where the source code is examined for errors and the practical rationality of the code is ensured [10]. The current code review process is problematic due to the large volume of code. It requires a significant amount of effort from the reviewers, and the traditional code review tools lack depth, failing to provide more in-depth feedback. Based on this background, there has emerged a trend of using LLMs for automated code review. The core value of LLM in code review lies in its powerful semantic understanding and context reasoning capabilities, enabling it to surpass traditional rule-based static analysis tools [5]. This

section will demonstrate the specific implementation and application of LLM in code review by taking Copilot as an example.

2.2.1 Copilot

2.2.1.1 The Application Focuses on the Operational Mechanism

GitHub Copilot, as a pioneer in the field of AI-assisted programming, its code review function demonstrates the potential of LLM in this area [18].

The operation mechanism of GitHub Copilot in code review mainly involves analyzing the code blocks or modifications submitted by developers in the integrated development environment (IDE) in real time, and providing feedback in the form of embedded comments, suggested completions, or differentiated suggestions [18]. It can not only automatically suggest a uniform code style, variable naming and comment format, but also identify and correct spelling errors, grammatical mistakes, as well as simple structural adjustments that may affect the clarity of the code [18].

2.2.1.1.1 Case Study: Unsafe Function Usage Warning

Consider a backend developer writing a function in C to handle user input, utilizing a traditional, unsafe string copying function (shown in Figure 1).

```
1  void copy_data(char *destination, const char *source) {
2      // Copilot Real-time Review Starts
3      // Uses a dangerous function without length checking
4      strcpy(destination, source);
5  }
```

Figure 1. Copilot Code Review Case

Immediately after the developer types *strcpy*, Copilot might trigger a real-time warning, indicating that this function poses a Buffer Overflow security risk because it does not check the size of the destination buffer.

Then, Copilot suggests using a safer alternative, such as *strncpy* or *strcpy_s*, emphasizing that the buffer size must always be passed as a parameter. This embedded real-time suggestion helps developers integrate memory safety awareness directly into the coding phase.

2.2.1.2 The Challenges of Detecting Critical Security Flaws

Although LLM has a strong ability for semantic understanding, it also has some limitations when it comes to detecting critical security vulnerabilities [5]. A study systematically evaluated the effectiveness of Copilot's code review function in detecting security flaws. The results showed that it often failed to detect critical vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), and insecure deserialization [18].

This incident highlights the limitations of current AI-assisted code review: The model excels in ensuring code "readability" and "conformity", but still cannot replace dedicated security tools and manual review in areas that require deep security knowledge and complex data flow analysis [5].

3. Discussion

Table 2. The Comparison of AI-Assisted Code Generation and AI-Driven Code Review

Comparison Metric	AI-Assisted Code Generation	AI-Driven Code Review
Primary Objective	Enhance Developer Productivity and accelerate delivery speed. Automate code writing to reduce repetitive work.	Ensure Code Quality and proactive defect detection. Reduce human effort and broaden the scope of issue detection.
Advantages	1. Strong Instruction Following: Can convert natural language requirements into executable code (Code Llama [14]). 2. Long Context Dependency: Supports understanding complex project-level contexts and dependencies. 3. Real-Time Infilling: Embedded tools like Copilot [15] provide instant code completion, significantly boosting coding speed.	1. Real-Time Embedded Feedback: Instantly identifies and corrects code style, idioms, and low-severity defects within the IDE [18]. 2. Semantic Understanding: Can identify Code Smells and potential future risks that traditional tools often miss [13]. 3. Reduced Human Cost: Effectively handles a large volume of low-level, repetitive review tasks, allowing human focus on core logic.
Disadvantages	1. Reliability Risk: Generated code correctness rate is moderate (Copilot approx. 46.3% [17]). 2. Security Risk: Prone to generating code with security vulnerabilities or unsafe API usage [18]. 3. Verification Cost: Developers must spend extra time verifying and refactoring the generated code.	1. Inadequate High-Risk Vulnerability Detection: Struggles to find complex, data-flow-dependent security flaws like SQL Injection [18]. 2. Lack of Explainability: Review suggestions lack underlying reasoning, affecting developer trust and adoption.
Future Development Focus	1. Agent-Based Closed-Loop Generation: Developing AI Agent models capable of self-testing and self-correction. 2. Quality Improvement: Focusing on generating high-reliability, high-security code to reduce post-verification costs. 3. Customization: Enhancing fine-tuning capabilities for specific enterprise architectures and domain languages.	1. Multimodal Analysis: Combining code, test reports, and architectural diagrams for more robust defect prediction. 2. Explainable Security Review: Improving security vulnerability detection and providing detailed Root Cause Analysis. 3. Human-AI Collaboration Optimization: Exploring seamless integration of LLMs into the human review process to boost overall efficiency.

4. Conclusion

In the field of code generation, LLMs such as Code Llama and GitHub Copilot have become indispensable tools for developers in their daily work. By leveraging long sequence context and the ability to follow natural language instructions, they achieve rapid conversion from requirement descriptions to function/module skeletons, significantly accelerating the code delivery speed. On the other hand, in code review, LLMs demonstrate strong semantic understanding and context reasoning capabilities, effectively identifying code anomalies, potential bugs, and providing best practice suggestions. Tools like Copilot significantly reduce the burden on human reviewers by automating formatting and handling low-severity issues.

Although AI has achieved remarkable results in enhancing efficiency and ensuring initial quality, it still faces multiple significant challenges in the process of moving towards a fully automated intelligent development paradigm.

Although the code generated by LLM has high grammatical validity, its logical correctness still needs to be improved. For instance, the evaluation results show that the proportion of the code generated by Copilot that is logically correct is only approximately 46.3%. This indicates that developers still need to invest a lot of effort in verification and debugging. In terms of code review, LLM-assisted tools (such as Copilot) perform poorly in detecting high-severity security

vulnerabilities (such as SQL injection, XSS, and insecure deserialization). This reveals the limitations of LLM in complex data flow analysis and deep security knowledge, and it cannot replace dedicated security tools and manual audits.

Future research on AI-driven software development needs to combine the semantic understanding advantages of LLM with the accuracy of traditional static analysis tools to build hybrid AI Agents. These Agents will focus on security sensitivity and performance optimization, and be capable of performing complex program analysis (such as data flow and control flow analysis), in order to overcome the limitations of current LLM in detecting high-risk defects. It is also necessary to further fine-tune LLM for specific backend technology stacks and enterprise internal code repositories to enhance the model's ability to understand complex business logic and architectural specifications, and improve the domain relevance and accuracy of the generated code.

References

[1] Nyaga, F. (2025). AI-Driven Software Engineering: A Systematic Review of Machine Learning's Impact and Future Directions. Preprints. https://doi.org/10.20944/preprints202504.0174.v1.

[2] Konakanchi, S. (2025). Artificial Intelligence in Code Optimization and Refactoring. Journal of Data and Digital Innovation (JDDI), 2(1), 9-35.

[3] Rao, B. S. M., Bandari, S. S. G., & Nc, R. (2025). Replacing AI Agents for Backend.

[4] Fang, C., Miao, N., Srivastav, S., Liu, J., Zhang, R., Fang, R., ... & Homayoun, H. (2024). Large language models for code analysis: Do {LLMs} really do their job?. In 33rd USENIX Security Symposium (USENIX Security 24) (pp. 829-846).

[5] Konda, R. AI-Powered Code Review Enhancing Software Quality with Intelligent Agents. IJLRP-International Journal of Leading Research Publication, 4(3).

[6] Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., & Myers, B. (2024, April). Using an llm to help with code understanding. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (pp. 1-13).

[7] Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., ... & Ma, Y. (2024). Exploring and evaluating hallucinations in llm-powered code generation. arXiv preprint arXiv:2404.00971.

[8] Crawford, T., Duong, S., Fueston, R., Lawani, A., Owoade, S., Uzoka, A., ... & Yazdinejad, A. (2023). AI in software engineering: a survey on project management applications. arXiv preprint arXiv: 2307. 15224.

[9] Agha, A. S. (2025). Evaluating AI Efficiency in Backend Software Development-A Comparative Analysis Across Frameworks.

[10] Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K. K., Wang, X., Nguyen, A., ... & Abrahamsson, P. (2024). Ai-powered code review with llms: Early results. arXiv preprint arXiv:2404.18496.

[11] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[12] Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., ... & de Vries, H. (2023). Starcoder: may the source be with you!. arXiv preprint arXiv:2305.06161.

[13] Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., ... & Lewis, M. (2022). Incoder: A generative model for code infilling and synthesis. arXiv preprint arXiv:2204.05999. p.

[14] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., ... & Synnaeve, G. (2023). Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

[15] Yetistiren, B., Ozsoy, I., & Tuzun, E. (2022, November). Assessing the quality of GitHub copilot's code generation. In Proceedings of the 18th international conference on predictive models and data analytics in software engineering (pp. 62-71).

[16] Dong, Y., Jiang, X., Qian, J., Wang, T., Zhang, K., Jin, Z., & Li, G. (2025). A survey on code generation with llm-based agents. arXiv preprint arXiv:2508.00083.

[17] Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. arXiv preprint arXiv:2304.10778.

[18] Amro, A., & Alalfi, M. H. (2025). GitHub's Copilot Code Review: Can AI Spot Security Flaws Before You Commit?. arXiv preprint arXiv:2509.13650.