Vol. 16 (2026): Proceedings of the 2026 International Conference on AI and Technological Innovation Applications (AITIA 2026)
E-ISSN: 3105-0522 P-ISSN: 3105-0514
DOI: https://doi.org/10.70267/aitia.2026182187

From Typist to Collaborator: A Systematic Review of AI's Impact on Programming Efficiency and the Shift in Developer Roles

Ziyu Wang*

Qingdao University, Qingdao, China

**Corresponding author: Ziyu Wang.*

Abstract

With continuous advancements in AI and large language models, AI in programming has evolved from simple code completion to intelligent agents with contextual understanding. This change has not only had a significant impact on the field of programming but also on programmers' employment. This paper aims to explore new working models for future programmers and analyze the efficiency and potential risks of AI programming. This paper systematically reviews key empirical studies from relevant papers published between 2020 and 2025 and finds that AI exhibits significant efficiency issues in programming. On the one hand, in LeetCode, AI performs well on low-to-medium difficulty problems, and AI programming has already surpassed 85% of human programmers in algorithm competitions. On the other hand, the actual development efficiency of AI when handling large open-source projects actually decreases by approximately 19%, and a series of safety and efficiency issues are found in AI programming, such as code redundancy and over-coupling when writing large projects. This article explores the transformation of programmers from code writers to AI collaborators, examining the advantages and disadvantages of AI in programming. It also elaborates on the specific requirements for future programmers, such as a deeper understanding of the nature of tasks and the need to maintain a dynamic balance between programmers and AI to achieve greater efficiency in project completion.

Keywords

artificial intelligence, programmer, project development

1. Introduction

Today's AI is vastly different from the AI that was just released in ChatGPT-3 in 2020. AI is becoming increasingly sophisticated, especially in programming, and a series of test datasets has emerged to evaluate the programming capabilities of large language models [1]. Moreover, in March 2026, Openclaw (an open-source, local-first AI execution hub) broke through 250K+ stars on GitHub, surpassing React (a JavaScript library for building user interfaces, mainly used for developing web and mobile applications), which had long held the top spot, and became the software project with the most stars on GitHub [12]. How programmers should change their work patterns in the future and how to maximize the effectiveness of AI programming have become very necessary issues to discuss. First, AI has a huge impact on the field of programming. Currently, AI

programming can achieve good results on common algorithmic problems, even surpassing many students and professionals with programming experience [2][3]. Moreover, some large language models can even spontaneously recover from error states and avoid program crashes during long-term operation [4]. Secondly, a series of potential hidden dangers and efficiency problems have been shown in AI in large-scale projects. Experiments have shown that AI tends to add more code, which, in turn, leads to many code redundancies and coupling problems [5]. At the same time, when doing large-scale projects, the efficiency of AI tools is significantly lower than expected [6]. Thirdly, many documents also emphasize that humans should change from typists to supervisors, thereby adding a layer of protection to AI code [7]. However, the existing literature does not specify the kind of working mode we should adopt to adapt to the current era of rapid AI development. Therefore, it is difficult for us to find a balance point that allows us to fully realize AI's potential. Therefore, this paper explores a working mode suitable for programmers now and in the future, as well as the current and future working status and division of labor between programmers and AI, by comprehensively analyzing the literature and empirical evidence. In this way, the work efficiency and output quality of programmers when undertaking large-scale projects can be significantly improved now and in the future. It can also provide new directions for future training of large language models and solutions, as well as new ideas for potential problems in AI programming.

2. Theoretical Background

The Human-AI Collaboration and Task Partitioning Theory was systematically proposed by Nguyen Duc and others (2025), and its core is the rational allocation of tasks based on the respective capability boundaries of humans and artificial intelligence. The theory posits that humans and AI have complementary advantages: AI excels at handling local tasks with high determinacy, strong repetitiveness, and limited context dependency; whereas humans are more advantageous in high-complexity, highly dependent scenarios that require a global perspective, ethical judgment, and cross-domain communication [7]. Transaction Cost Theory: The core conclusion of this theory is that, although artificial intelligence can significantly reduce code-generation costs (approaching zero in some scenarios), verification costs increase exponentially with project scale and complexity. This cost asymmetry can explain the efficiency paradox observed in empirical studies: artificial intelligence performs well on medium- and low-difficulty tasks (where verification costs are low), but in large projects it can actually decrease efficiency (where verification costs outweigh the benefits of generation)[6].

3. Literature Review

3.1 The Impact of AI Tools on Project Development

AI-assisted tools have completely changed the landscape of software development. Testing, debugging, analysis, optimization, and even code generation can be achieved with AI tools. On LeetCode (an online coding platform), ChatGPT3.5's programming ability was tested on 180 programming problems. The experimental results showed that ChatGPT3.5 was more effective than human code on simple and medium difficulty problems. Still, the reliability of code generated by ChatGPT3.5 was low when solving difficult problems [2]. The failure of AI to handle long, complex problems like those on LeetCode's difficult problems stems from its autoregressive, generative nature. Because LLM (Limited Logic Model) constructs code by predicting the next token, the correctness of each step in the problem-solving process depends on the accuracy of previous tokens. In long logic tasks, the tiny logical shifts introduced by LLM are amplified during the reasoning process, eventually leading to serious logical errors. This “illusory drift” prevents the model from maintaining global consistency across long algorithms that require rigorous causal reasoning, leading to a significant decrease in reliability on LeetCode's difficult problems. However, with the continuous updates and iterations of Large Language Models (LLMs), some AI tools can run continuously for several hours while maintaining task consistency, avoiding deadlocks, and recovering from errors, such as Anthropic's Claude Opus 4 [4]. Therefore, to test the performance of LLMs in programming, many code-generation benchmarking tests at the code library level have emerged, such as RepoBench, LongCode-Arena, and DevEval [1]. And the development of large projects is largely affected by prompt words. Prompt word engineering is not only natural language input, but also a complex, iterative 'design search' process [8]. Therefore, under this dual update, AI programming capabilities have made a huge leap forward.

3.2 The Difference Between AI Code and Human Code

Although AI programming is becoming increasingly powerful, most internet companies worldwide have not yet experienced large-scale programmer layoffs due to AI, suggesting that there is indeed a difference between AI code and human code.

Research shows that AI programming tends to “add code” to solve problems, rather than simplifying code through “refactoring” and “introducing abstractions” like senior engineers do [5]. This indirectly proves why AI-generated code easily becomes “ code bloat and poor maintainability “ of code [5]. And because of the limitations of “Context Window”, the model has a strong perception of the beginning and end of the input text, but it is prone to losing complex dependencies in the middle. When handling large projects, because project architectures often involve deeply nested class calls and complex, decoupled interfaces, these key pieces of information often exceed the model's effective attention span. When the model cannot anchor both the 'underlying implementation' and the 'high-level calls' within a limited token window, it tends to generate a set of self-consistent but redundant local logic instead of calling existing global interfaces, which directly leads to high coupling and fragmented code. Moreover, since AI models are trained on a large amount of unapproved open-source code, AI inevitably learns the mistakes that human developers have made [9]. Therefore, nearly 40% of AI-generated code contains security vulnerabilities [9]. In the nearly 5 years since AI was developed, the ChatGPT4 model, under an optimal prompt-word strategy, outperforms 85% of human contestants (most of whom are students and professionals with intermediate programming experience)[3]. Moreover, AI performs exceptionally well in “cross-language conversion” (e.g., from Python to C++)[3]. Therefore, it is not difficult to conclude that with the recent release of GPT5.4 and the emergence of AI agents such as OpenClaw, this number will be even higher. Although Pearce et al. warned of the security deficiencies of AI-generated code, the latest research by Hou et al. revealed the terrifying, explosive power of AI in logical implementation. In the LeetCode competition, top models have already defeated 85% of human developers. This means that AI-generated code has gradually closed the gap with human-generated code in terms of algorithmic efficiency. Therefore, the challenge facing contemporary programmers is no longer “how to write faster algorithms,” but rather how to use the “intent-driven” model proposed by Hassan (2026) to guide AI in quickly producing high-performance system architectures while ensuring security.

3.3 Changes in the Work of Programmers

We cannot ignore the impact of AI on the development field, and a considerable number of programmers have already used AI (such as Claude opus) as a powerful code-writing assistant. Generative AI has become an indispensable tool for programmers. As AI undertakes more coding tasks, programmers' work will involve greater communication, coordination, and decision-making. The role of software engineers is shifting from simple code writers to system integrators and AI supervisors [7]. However, the time spent using and supervising AI will also affect programmers' efficiency in completing their work. For example, a randomized controlled trial (RCT) for senior open source developers. The study included 16 developers with an average of 5 years of project experience, who completed 246 tasks and compared the use of the most powerful AI tools (such as Cursor Pro, Claude 3.5/3.7 Sonnet) in early 2025, when AI use was prohibited [8]. Surprisingly, the study found that allowing the use of AI actually increased the task completion time by 19% - AI tools actually slowed down the progress of senior developers [6]. Moreover, despite the actual decrease in efficiency, developers still subjectively believe that AI has shortened the time by 20%. This “disconnect between perception and reality” is a significant feature of AI-assisted development in 2025 [6].

Therefore, we face an important question: how should we use AI to improve our work efficiency, or even reach its peak? Regarding the issue of decreased efficiency in AI-generated tasks mentioned above, although AI can complete most repetitive, simple code, the time programmers spend correcting and reviewing the AI-generated code exceeds the time they spend writing the code themselves. This explains why efficiency actually decreases with AI. In the future, the core competitiveness of programmers will no longer be about the speed of logic implementation, but rather about transforming from “typists” to “collaborators” to maximize the power of AI tools. For example, let AI generate code first, and when problems occur, programmers will refactor the commands and prompts, and finally perform manual fine-tuning themselves. Secondly, programmers should also be able to determine whether the task type is suitable for AI to perform. This also requires programmers to have a better understanding of the project's future situation. In general, programmers should learn to maintain a dynamic balance with AI. In other words, programmers must find a balance between over-reliance

(leading to overlooking security vulnerabilities) and over-skepticism (leading to repetitive manual verification). If trust levels are unbalanced, programmers will fall into a "cognitive deadlock," spending excessive time speculating about the non-linear logic of AI rather than proactively building the system. On the other hand, we must examine the "hidden costs" of AI. AI lacks an understanding of the overall project lifecycle, and the code it generates is often poorly maintainable. This pursuit of local optima can easily escalate into heavy technical debt in the later stages of a project. Therefore, future high-efficiency developers must not only possess the ability to command AI but also the foresight to examine the long-term evolution of code. Only when programmers can identify and block the fragmented logic introduced by AI can AI truly be transformed into a genuine "productivity lever."

4. Discussion

4.1 Programmers' Attitudes Towards AI

In real life, AI efficiency is affected by many factors, including programmers' own optimism or lack of trust in AI. This "distrust" will cause programmers to conduct overly detailed reviews of AI-generated code and even rewrite it [10]. For example, programmers spend a lot of time writing prompts, waiting for them to be generated, and debugging them. This ongoing doubt about the quality of AI output forces developers to invest significant energy in verification [10]. The decline in efficiency will further exacerbate programmers' distrust of AI, forming a vicious cycle. This is why this paper encourages programmers to regard themselves as "collaborators". On the one hand, it will reduce the psychological burden on programmers; on the other hand, it will promote programmers' understanding of AI, making AI better suited to the project.

4.2 The "Fragmentation" Problem of AI Code

As mentioned earlier, code generated by AI often leads to serious "technical debt" due to excessive code additions [5]. Although AI is very good at solving local problems, it lacks a global architectural perspective. This is also a major reason why AI's efficiency in experiments actually declines. This is also a key point that programmers should pay attention to in the future: how to allocate their energy to address problems at work in a new way. According to various literature, programmers should devote 60% of their energy to the precise description of requirements (such as writing prompts) and formal expression, and use the remaining 40% to address security risks and "technical debt" in code generated by AI. This ratio will also be updated as AI tools continue to evolve. For example, if the problem of "technical debt" is effectively solved in the future, then the energy used to deal with AI code can be reduced.

4.3 On the Transformation of Future Programmer Requirements

The literature predicts that future programming will become a "high-precision specification" behavior, meaning that programmers' core work will be to define the constraints, boundaries, and business logic of the system [11]. This echoes the core view of this paper, the identity of programmers as "collaborators". In other words, in the future, the key value of programmers lies in their ability to understand "why to do it" better than AI. At the same time, it can also allow programmers to return to studying underlying principles (such as memory management and complexity analysis), thereby better solving deep bugs in specific problems. Ultimately, this "collaborator" model will form a closed loop: programmers are responsible for defining boundaries and handling high-entropy business decisions, while AI is responsible for low-entropy repetitive implementation. This division of labor will force programmers to return from the "code production line" to the essence of "systems engineering," thereby gaining the ultimate decision-making power to address the collapse of complex systems by understanding "why it is done this way."

5. Conclusion

This paper, by repeatedly extracting the core arguments and research results from several relevant papers and comparing the current level of AI programming with that of past AI, points out that future programmers and AI should maintain a dynamically balanced working relationship in large-scale projects. This means treating AI as a colleague and collaborator, rather than simply having programmers act as typists or supervisors. Simultaneously, the programmer's core responsibilities are changing as AI develops. Programmers are required

to have a deeper understanding of their projects, going beyond simply solving algorithmic exercises to grasp the project's purpose, background, and nature, and to make a preliminary judgment on whether the project is suitable for AI. While this paper elaborates on the changing work patterns of future programmers and new approaches to using AI tools, the issue of unclear division of labor remains. For example, in large-scale projects, which tasks are suitable for AI and which require programmer review? Future research should delve deeper into the specific task division within a project team for large-scale projects, summarizing and categorizing each task, and endowing them with new properties under an AI-human programming model. This paper initially proposes using task complexity as a defining criterion; for example, tasks with high determinism and reusability should be handled by AI, while tasks involving cross-domain communication, ethical decision-making, and blurred business boundaries should be handled primarily by human programmers. This paper does not simply define the programmer's role as a supervisor, but rather considers the current programming level of AI (e.g., Claude Opus 4), its development speed (e.g., ChatGPT3.5 to ChatGPT5.4 took only about four years), and future trends (OpenClaw's star count ranks first among software projects) to arrive at a new conclusion about "collaborators," providing a reliable approach for future AI research and training.

References

[1] Liang, S., Jiang, N., Hu, Y., & Tan, L. (2025). *Can Language Models Replace Programmers for Coding?*. Association for Computational Linguistics.

[2] Kuhail, M. A., Mathew, S. S., Khalil, A., Berengueres, J., & Shah, S. J. H. (2024). *"Will I be replaced?" Assessing ChatGPT's effect on software development and programmer perceptions of AI tools*. Elsevier.

[3] Hou, W., & Ji, Z. (2024). *Comparing large language models and human programmers for generating programming code*. Wiley.

[4] Wang, H., Gong, J., Zhang, H., Xu, J., & Wang, Z. (2024). *AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities*. Association for Computing Machinery.

[5] Hassan, A. E., Oliva, G. A., Lin, D., Chen, B., & Jiang, Z. M. (2024). *Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap*. arXiv.

[6] Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity*. arXiv.

[7] Nguyen-Duc, A., Cabrero-Daniel, B., Przybylek, A., Arora, C., Khanna, D., Herda, T., Rafiq, U., Melegati, J., Guerra, E., Kemell, K.-K., Saari, M., Zhang, Z., Le, H., Quan, T., & Abrahamsson, P. (2025). *Generative Artificial Intelligence for Software Engineering—A Research Agenda*. Wiley.

[8] Liu, V., & Chilton, L. B. (2022). *Design Guidelines for Prompt Engineering Text-to-Image Generative Models*. Association for Computing Machinery.

[9] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2025). *Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions*. Association for Computing Machinery.

[10] Weisz, J. D., Kumar, S. V., Muller, M., Borromeo, K. E., Goldberg, A., Heintze, K. E., & Bajpa, S. (2025). *Examining the Use and Impact of an AI Code Assistant on Developer Productivity and Experience in the Enterprise*. Association for Computing Machinery.

[11] Caballar, R. D. (2020). *Programming Without Code: The Rise of No-Code Software Development*. Institute of Electrical and Electronics Engineers.

[12] Digital Life Kazik. (2026, March 3). *We have witnessed history once again*. WeChat Official Account. https://mp.weixin.qq.com/s/MTnpLe4KxTU0B9PgKa4Yyw

Funding

This research received no external funding.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgment

This paper is an output of the science project.