

Reflections on the Impact Brought by Large Language Models to Software Engineering Maintenance Modes Under the Wave of Artificial Intelligence

Wenqian Zhang

Faculty of Computer Science and Information Technology, Universiti Malaya, Kuala Lumpur, 50603, Malaysia

Abstract: Software maintenance occurs throughout the life cycle of the system and is relatively expensive. The Introduction of large language models is changing how software maintenance is conducted. The five areas this paper investigates for the practical applications of large language models in software maintenance are: maintenance process, collaborative verification, legacy system transformation, maintenance cognition, and security governance. According to a survey of almost 5,000 technical professionals in 2025 by the DORA report, 90% of the practitioners have already used AI tools in their work, and more than 80% believe that productivity has increased. AI is only an "enhancer" of existing delivery modes; thus, only organisations with well-established platforms and processes will reap significant benefits, and otherwise, technical debt will accumulate at a high rate. According to the qualitative analysis of DORA, most of the time saved in the code generation stage is spent on review and verification. Therefore, the focus of maintenance work has shifted from writing to judgment, and the reliability of AI modifications is now more dependent on system-level understanding than on manual modification. Large Language Models have not solved the problem of maintenance; they have only moved it from the coding stage to the stages of verification, governance and architecture design.

Keywords: Large Language Models, Software Maintenance, Legacy Systems, Human-Machine Collaboration, Code Review, Technical Debt.

1. Introduction

Software maintenance is the most resource-intensive part of software engineering. Several studies by the IEEE technical committee have shown that maintenance expenses generally account for 60%-80% of the total cost of ownership for software. Brooks, in his 1987 paper "No Silver Bullet", had already indicated that due to the inherent complexity of software, no single technological innovation would be able to increase productivity tenfold within ten years. Although the development tools and programming languages have changed constantly over the past forty years, the basic process of maintenance work has not changed: read the existing code, understand its logic, find errors, make modifications, and verify the results. The development of large language models has begun to change this situation.

According to the 2025 DORA report, a survey of nearly 5,000 technical professionals around the world, many are already using AI tools in their work, and more than 80% of them believe that AI has enhanced their personal productivity [1]. However, the report also shows that the effect of AI on the efficiency of software delivery is not a uniform improvement; rather, it acts as an "amplifier": when organisational platforms are mature and processes are clear, AI is an effective collaboration tool; when tools are scattered and infrastructure is weak, AI will accelerate the accumulation of technical debt. Qualitative analysis of DORA also found that most of the time saved in code generation was moved to review and verification. Maintenance work does not simply mean "fixing bugs faster"; it refers to a change in the direction of work, as the reliability of AI modifications is now relatively low and highly dependent on an understanding of the entire system.

This leads to the main question of this paper: To what

extent have large language models altered the actual model of software maintenance, which modifications are the most pronounced, and what new problems have arisen due to these changes? Around these problems, the five levels in the article are: maintenance process, human-machine collaboration review, legacy system modernisation, maintenance cognition reconstruction, and security governance, and all arguments are based on verifiable academic literature and industry reports.

2. The Structural Reshaping of Maintenance Processes by Large Language Models

The three types of traditional software maintenance are understanding, localisation and modification; the understanding stage is relatively time-consuming. The developer needs to read the source code, trace the call relationships, and refer to the documentation to find the modification location. Large Language Models are changing the starting point and direction of this road ahead [2].

In the code understanding and change location stages of the code, intelligent agent tools based on retrieval-augmented generation (RAG) technology can automatically retrieve relevant context from the code repository and generate change impact analysis. According to relevant review literature, agent methods based on large language models have recently attracted a large number of studies in the area of software problem-solving. These systems have moved from single-step generation to multi-step reasoning and iterative exploration, and are now at the forefront of automated software maintenance. Developers do not need to track code dependencies manually anymore. They can state the maintenance requirements in natural language, and then the

model will help position it initially.

A change in the focus of the process has had an uneven effect; not all areas have been spared. According to the thematic analysis of 1,110 open-ended questionnaire responses from Google engineers in the DORA 2025 report, most of the time saved by AI in the code generation stage was moved to the review and verification stage. Therefore, the work rhythm in the maintenance scenario has changed. In the past, a considerable amount of time was spent learning the code, and only a small fraction was used for writing modifications. Now, it is difficult to obtain modification suggestions promptly, and a lot of time is being wasted determining the credibility of these suggestions. Therefore, the skills required of maintenance staff will also change. A system-level comprehension of safety cannot be achieved by hand-coding to determine whether the AI-generated modification is safe.

In addition to changes in the focus and demands of skills, the scale of maintenance work is also growing. Traditionally, maintenance is often carried out for a single defect or minor modification. With the help of large language models, maintenance activities are now extending to both ends. On one end is fine-grained single-line completion and instant repair suggestions, and on the other end is a large-scale repository-level modification [3]. In November 2025, the Spotify engineering team posted on the engineering blog that the background coding agent had successfully managed a large-scale code migration of several thousand software modules and generated more than 1,500 merged pull requests. The former large-scale maintenance needed specialised abstract syntax tree operation scripts and deep infrastructure knowledge. It can now be operated using natural language instructions.

3. Code Review and Maintenance Verification Under Human-Machine Collaboration

The real problem of software maintenance is not "inability to modify", but "modifying the wrong thing". A small change is still shown in the regression error; therefore, this problem has not been solved. With the arrival of large language models, the subsequent maintenance and verification work are now more challenging. Previously, the developer only needed to ensure the correctness of the modifications. First, check if the proposed solutions of the model are practical in the current situation. Therefore, it is a new kind of verification liability.

In October 2025, GitHub Copilot has improved the code review function in an update. At present, it is no longer only a way to explore LLMs, but rather integrates the semantic reasoning of LLMs with a deterministic rule engine and employs static analysis tools such as CodeQL and ESLint, along with model judgments. Therefore, it can be seen that only semantic understanding of large language models is insufficient for dependable code inspection. A verifiable deterministic analysis is also used for backup. The choice of a hybrid architecture is also a practical one; although AI can assist in decision-making, the final guarantee of safety still needs to be provided by reproducible rules.

The actual results of the hybrid architecture have been presented in some recent studies. A study presented at the ESEM conference in 2025 compared LLM-generated code with human-written code using SonarQube. Based on the above results, LLM-generated code has fewer bugs and lower

repair costs in general; however, in complex problem scenarios at the competition level, the LLM solution introduced structural defects that were absent in human code [4]. In other words, AI-generated code performs well for routine maintenance; however, in the presence of a complex architecture and non-functional requirements, the reliability of the AI's suggestions drops considerably. The maintenance team will decide whether to use artificial intelligence or a person for the operation at the border line.

In addition to the reliability problems of the review process, the accumulation of technical debt is also a maintenance verification problem. According to the research of the ATeam framework, AI-assisted development in a vibe coding style does accelerate code generation; however, in an enterprise maintenance environment, it may introduce security vulnerabilities, accumulate technical debt, and even result in architectural deviations. The researchers of the ATeam framework carried out an evaluation of 24 sustaining engineering tasks and found that structured decomposition and governance mechanisms are the key to reliable maintenance; the size of the model itself is not very important—even small models can be competitive under a good governance system [5]. The switch to maintenance mode should not be regarded as an upgrade of an all-powerful model; rather, changes are also needed in the process and governance.

4. Modernization Practice of Legacy Systems Driven by Large Language Models

Maintenance of legacy systems is very difficult in software engineering. The programming languages of the system are outdated, there is no documentation, the original developers have left a long time ago, but the business cannot operate without it. The impact of a large language model on this area will determine the extent to which the maintenance model is altered.

The SCORE system of Volvo Group is a typical case. In 2017, the system was introduced for the use of the HR department and managers to manage organisational structure, store employee information and handle operational workflows. Later, it showed poor performance and was slow in the database and interface. Volvo employed GPT-4 and Claude AI to re-engineer the system in three ways: zero-shot, one-shot and few-shot. The API response time has decreased from 20-30 seconds to 3 seconds, and at the same time, database queries and code modularisation have also been optimised. This case study has demonstrated that large language models can achieve tangible results in the digitalisation of old systems; otherwise, they would only be at the level of ideas.

MITRE's IT Modernisation Pipeline is another way. They placed the legacy software in a secure environment with a private LLM, and, using self-developed tools and context-aware prompt generators, performed semi-automated code translation and modernisation. Technically, MITRE used Tree-sitter to construct specific syntax trees, divided the code into logical sections according to functional blocks to adapt to the context window of the model, and then employed multiple-choice quizzes to have the LLM itself identify the areas that required emphasis in translation [6]. According to MITRE, LLMs should be employed under strict supervision for legacy modernisation of mission-critical systems.

Another one of the details of MITRE's work that people often forget is context management. In 2026, ICSA-C published a study proposing the construction of a structured knowledge graph based on parsing for multi-hop architecture reasoning in legacy systems using GraphRAG, and it showed improved retrieval accuracy compared with the pure text baseline [7]. This suggests that the effect of LLMs on legacy systems depends largely on the context they are given; the quality of model output for unstructured source code text is significantly different from that of structured input after parsing and knowledge graph construction. For maintenance practices, this means that when introducing LLMs into the maintenance of legacy systems, instead of investing resources in upgrading model versions, it is better to first structure the code and context.

5. Cognitive Reconstruction and Capability Migration of Maintenance Roles

The most profound impact of large language models on the maintenance model may not be an improvement in the efficiency at the tool level, but rather a change in the basic concept of "what maintenance work is".

The original function of developers in old-school software maintenance was to understand the code and make small changes. They need to construct an execution model of the code in their minds, understand how the state of variables changes, and predict the chain reactions caused by a modification. This capability gradually arises with the accumulation of experience in a particular code base. With the introduction of large language models, some of the problems of "understanding code" have been solved. Based on the above research, as large language models become more integrated into software development, the role of programmers is changing from that of traditional coders to that of strategic partners; their work will shift from writing code line by line to ensuring the accuracy and security of AI-generated results. Therefore, developers need to spend more effort verifying the correctness and security of AI-generated code; as a result of this role change, there is a "de-codification" of maintenance capabilities [8].

The main ability of maintenance work has shifted from "quickly and accurately reading and writing code" to "accurately determining the reliability of AI-generated modifications". Maintenance Staff need System Thinking and verification skills, not fast typing. As shown in the ESEM 2025 study, the fine-tuned model reduced the number of high-severity problems but lowered the general performance. Do not skip manual verification based on the assurance provided by AI. The new demands for the maintenance team's capabilities are also leading to changes in its organisation. A reference case is derived from the internal operations of Tencent.

Internal practice data of Tencent Cloud AI Code Assistant in the Tencent Medical Health team show that after almost a year of use, the team achieved 100% coverage and maintained active use. The weekly code completion rate was 39.81% and the weekly adoption rate was 31.63% [9]. Given the scale of deployment, the organisation's method for carrying out maintenance work should also be adjusted; for instance, incorporate the results of AI review into the code review process, consider the specific characteristics of AI-generated code when assigning maintenance tasks, and distinguish the

defect types of manual code from those of AI code in the quality measurement system. The organisation's modification efforts are more difficult to achieve than introducing tools.

6. New Challenges in Maintenance from the Perspective of Security and Governance

Large language models are being used in software maintenance, but there is a problem of delay in releasing new versions because new bugs have been introduced. Security risks in the maintenance phase are different from those in the initial development period. Maintenance modifications are made to an existing system, and a single incorrect modification can cause disruption of the online business, which has a more direct impact than defects in newly developed code. Furthermore, the maintenance scenario often lacks full regression test coverage, and after a week or two after going live, an error modification may not be discovered until that time; consequently, the cost of troubleshooting will be significantly higher than during development.

The problem of LLM hallucination in the context of maintenance is particularly serious. A comprehensive study of hallucinations in code modification scenarios was conducted at the IJCNLP conference in 2025, and it was found that about 50% of the code review opinions generated by LLMs and 20% of the generated submission messages contained hallucination elements [10]. In code review and maintenance suggestion scenarios, hallucinations may occur in the form of the model confidently identifying non-existent problems or failing to report serious defects. According to the above data, directly using the LLM's review results to make maintenance decisions is not risk-free.

Refactoring of code occurs frequently during maintenance operations and is also a relatively concentrated area of LLM reliability problems. A paper presented at the 2nd IEEE/ACM IDE Symposium in 2025 pointed out that LLM-driven code refactoring has an error tendency and a possibility of hallucination, and thus cannot be fully integrated into the automated refactoring pipeline. Manually intervene to verify the changes that have been made [11]. Therefore, in the actual maintenance process, LLM refactoring suggestions should be treated as "drafts" rather than "final solutions" and need to pass dual checks of automated testing and manual review.

The Construction of the governance system will determine whether the maintenance mode for large language models can be implemented. The team has created a multi-stage pipeline for impact analysis and set up an explicit approval process for all AI-assisted maintenance modifications to ensure traceability and auditability. In the environment of enterprise software maintenance, technical debt management and architectural integrity must be guaranteed in all modifications. The conclusion of the DORA report is also in line with this: the success of AI depends on the quality of the organisation's systems. Organisations with mature DevOps practices, clear workflows and strong platform capabilities are more likely to leverage AI for productivity gains; otherwise, AI will only increase the accumulation of technical debt in fragmented organisations.

7. Conclusion

The effect of large language models on software maintenance is not to have AI replace humans. At the same time, change the focus and structure of skills in governance.

Time saved in the generation stage is used for review and verification, and the efficiency bottleneck shifts from coding ability to assessing the reliability of AI modifications. AI-generated code generally has fewer bugs and lower maintenance costs for routine problems; however, according to the ESEM 2025 study, it can introduce structural defects in complex scenarios that are absent in human-written code, and maintenance teams cannot rely on prior experience with human code to solve these issues. Legacy system modernisation has shown good results and is exemplified by the Volvo Group case study. According to MITRE's practices, the effectiveness of context engineering and human supervision varies greatly depending on the quality of the environment. The other is security governance. Approximately half of the LLM-generated code reviews contain hallucinations, and according to the evaluation of the ATeam framework, a governance mechanism is more responsible for reliable maintenance than the scale of the model. These results offer some support for exploring the changes in maintenance models, but this study is also limited in terms of method and scope.

This study has some defects. The cited cases are mainly from the practices of a few leading enterprises, and their applicability to small and medium-sized teams has not been tested. LLM technology changes rapidly; therefore, some of the data may be outdated after an update to the model. At present, the evaluation of the governance system has only been conducted on controlled experiments, and its long-term effects in actual production environments have not yet been observed.

Future research can be advanced in two directions. First, construct a specialised maintenance plan and quality-measurement system for AI-generated code to differentiate its defect characteristics from those of human-written code. The other is to add context engineering and governance mechanisms to the standardisation of maintenance processes, and test their transferability with more industry cases. The first type of change in maintenance is the shift from "writing" to "judging", and only if the organisation's processes and governance are highly responsive can this be achieved.

References

- [1] DORA. (n.d.). Balancing AI tensions: Moving from AI adoption to effective SDLC use. <https://dora.dev/insights/balancing-ai-tensions/>
- [2] Lyu, M. R., Ray, B., Roychoudhury, A., Tan, S. H., & Thongtanunam, P. (2025). Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology*, 34(5), 1–33.
- [3] Ma, Y., & Liu, Y. (2025). Improving automated issue resolution via comprehensive repository exploration. In *ICLR 2025 Third Workshop on Deep Learning for Code*.
- [4] Santa Molison, A., Moraes, M., Melo, G., Santos, F., & Assuncao, W. K. (2025, October). Is LLM-generated code more maintainable & reliable than human-written code?. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 151–162). IEEE.
- [5] Vella, S., Ferworn, A., & Sharieh, M. (2026, July). ATeam: Governance-Aware LLM-Assisted Software Sustaining Engineering for Enterprise Systems. In *2026 6th International Conference on Electrical, Computer and Energy Technologies (ICECET)* (pp. 1–6). IEEE.
- [6] Diggs, C., Doyle, M., Madan, A., Scott, S., Escamilla, E., Zimmer, J., ... & Thaker, S. (2024). Leveraging LLMs for legacy code modernization: Challenges and opportunities for LLM-generated documentation. *arXiv preprint arXiv:2411.14971*.
- [7] Thillmann, H., Rumpe, B., & Biesdorf, A. (2026, June). Bridging formal syntax and LLM semantics: Extracting knowledge graphs for legacy code understanding. In *2026 IEEE 23rd International Conference on Software Architecture Companion (ICSA-C)* (pp. 457–462). IEEE.
- [8] De La Cruz, E., Le, H., Meduri, K., Nadella, G. S., & Gonaygunta, H. (2025). Redefining the programmer: Human-AI collaboration, LLMs, and security in modern software engineering. *Computers, Materials & Continua*, 85(2), 3569–3582.
- [9] CloudStudio. (2025, February 12). Tencent Health: 40% of our code is written by AI. Tencent Cloud Developer Community. <https://cloud.tencent.cn/developer/article/2495772>
- [10] Liu, C., Lin, H. Y., & Thongtanunam, P. (2025, December). Hallucinations in code change to natural language generation: Prevalence and evaluation of detection metrics. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics* (pp. 2538–2560).
- [11] Cordeiro, J., Noei, S., & Zou, Y. (2025, May). Llm-driven code refactoring: Opportunities and limitations. In *2025 IEEE/ACM Second IDE Workshop (IDE)* (pp. 32–36). IEEE.