ISSN:2790-1688 Volume-15-(2025)

Large Language Models in Software Engineering: Automation, Collaboration, and Challenges

Cheng Zhong

Cuiyuan Middle School, Shenzhen, China

3669821583@qq.com

Abstract. As artificial intelligence (AI) technology develops rapidly, large-scale pre-trained language models (LLMs) have become widely applied in software engineering. This has revolutionized software development and brought significant value to testing, debugging, operations, project management, and other areas. This article systematically reviews the latest research findings on LLMs empowered software engineering through literature analysis and comparative research. It also provides a detailed explanation of requirements analysis and modeling, automated code generation, intelligent debugging, and operations and maintenance, and discusses software quality assurance and team collaboration. The article also explores the challenges encountered in the practical application of LLMs, primarily security and privacy concerns, difficulties in understanding and controlling, and issues such as copyright ownership and legal liability. Besides, the article offers predictions for future trends. Neural-symbolic integration, human-computer collaboration, and the establishment of standards are key drivers of continued progress in software engineering in the era of LLMs. Despite its contributions, this study is limited by its reliance on existing literature and secondary data, highlighting the need for future longitudinal and quantitative research to assess the real-world impact of LLMs on software quality, security, and maintainability.

Keywords: Large Language Model (LLM), Software Engineering, Automated Code Generation, Software Quality Assurance (SQA), Human–AI Collaboration.

1. Introduction

With the rapid development of artificial intelligence, particularly the rise of large-scale pre-trained language models (LLMs), software engineering is undergoing a major transformation [1]. Intelligent tools are gradually reshaping software engineering processes. LLMs, with their powerful natural language understanding and generation capabilities, assist developers with code editing, debugging, and documentation, as well as project organization, requirements clarification, and knowledge management [1]. This evolution marks a shift in software development from a human-led approach to a more collaborative relationship between humans and intelligent systems. However, this technological shift presents both opportunities and challenges. While large models enhance productivity and creativity in software development, they also raise issues related to reliability, data security, privacy, and interpretability. Generated code may still contain logical or structural flaws, and the opaque reasoning process of LLMs makes ongoing oversight difficult.

This study, using a literature review and comparative analysis, explores how large models are reshaping software engineering. By analyzing their strengths and emerging challenges, it aims to provide a deeper understanding of how LLMs can enhance development efficiency and innovation, while also highlighting key issues such as security, transparency, and accountability, which are crucial for the sustainable and trustworthy development of software engineering.

2. LLMs Drive a Paradigm Change in Software Development

2.1 Requirements Analysis and Modeling

Traditional software engineering requirements analysis relies on in-depth communication between requirements engineers and users, and is presented in the form of documents or modeling tools. However, this process is costly and uncertain. The emergence of LLMs has ushered in a new paradigm

shift in requirements analysis and modeling [3]. LLMs leverage their natural language processing capabilities to transform unstructured user descriptions into formalized requirements or modeling elements. For example, researchers have developed a requirements conversion system using models such as GPT-4 [4]. Users simply enter "building an e-commerce system with a shopping cart, payment, and logistics query," and the model creates the necessary use case diagrams or UML sequence diagrams. This approach significantly reduces ambiguity in requirements and improves standardization in the early stages of development.

Furthermore, LLMs support continuous requirements elicitation and dynamic modeling. In agile development iterations, the constant evolution of requirements makes timely document modifications difficult with traditional methods. By semantically parsing user feedback and historical requirements, LLMs can automatically generate iterative requirements interpretations and even predict future business requirements. This improves development efficiency and traceability. Numerous academic experiments have demonstrated the effectiveness of LLMs in requirements modeling. Recent studies have shown that in domains such as finance and healthcare, requirements generated by large language models (LLMs) were evaluated as more aligned with stakeholder needs (+1.12) and demonstrated a trend toward higher completeness (+10.2%) compared to those written by human experts [5]. When integrated with visual modeling tools, LLMs can transform natural language descriptions into structured models, supporting the "natural language-to-model" paradigm and laying the groundwork for automated software engineering.

2.2 Automated Code Generation

Code generation is one of the most popular applications of LLMs. Traditional automated code generation uses rule engines or specialized DSLs (domain-specific languages), resulting in very limited functionality. LLMs, pre-trained on a large amount of open source code, acquire the ability to understand and generate general programming languages, enabling cross-language and cross-platform code generation[4]. Empirical studies suggest that developers using GitHub Copilot complete certain repetitive coding tasks, unit test generation, debugging, and pair programming up to 30-40 % faster than without such tools [6]. This performance boost significantly reduces boilerplate and repetitive work, making Copilot an effective assistant in routine software development tasks. Beyond simple code generation, LLMs can also perform "context-aware generation." For example, if a project already has its own code structure, they can automatically generate code that matches it and shares the same style, rather than generating incompatible code. Furthermore, the model can be integrated with TDD, generating test cases based on user needs and then generating code that meets these tests, improving correctness. However, the "illusion" problem of LLMs still exists, and the generated code may contain potential logical or syntactic errors. To address these issues, the industry has proposed a human-machine collaboration model: developers use prompt engineering to improve input quality, and static analysis and unit testing to verify the generated code. This makes the LLMs a "code collaborator" rather than a complete replacement.

2.3 Intelligent Debugging and Operations

Debugging and maintenance consume the largest amount of human resources and time throughout the entire software development process. Traditional debugging methods rely on programmers reading log files and checking code line by line, which is inefficient and prone to overlooking issues. LLMs, leveraging their powerful semantic understanding and pattern matching capabilities, are driving intelligent debugging and operations. During debugging, they can automatically analyze program logs and error stack traces to quickly identify the source of bugs. For example, researchers have developed a debugging assistant based on big data models that can match the semantics of the contextual logs associated with a Null Pointer Exception and the error itself, generating possible causes and corresponding fixes. Experimental evaluations indicate that integrating LLM-based assistants into debugging workflows can substantially reduce bug-locating time, with some case studies reporting improvements on the order of tens of percent. For instance, tools integrating real-

time debugging hints within IDEs allow programmers to receive instant suggestions during coding, thereby accelerating error diagnosis and resolution [7].

At the operations level, LLMs are widely used in AIOps scenarios. By analyzing massive amounts of monitoring data and log streams, they can automatically identify potential anomalies, generate alert summaries, and predict system bottlenecks. For example, Microsoft and Google use LLMs to automate operations and maintenance on their cloud platforms, reducing the time spent manually handling alerts. LLMs can also help generate operations and maintenance scripts, enabling self-healing and reducing the risk of downtime. However, this also presents operational challenges. With the sheer volume of system logs, models can misdiagnose and even generate incorrect repair solutions. Therefore, researchers have proposed a "human-machine collaborative closed loop," where the model first generates candidate solutions, which are then verified and deployed by operations engineers. This mechanism leverages the model's strengths while ensuring system security and stability.

3. Empowering Software Quality Assurance through Large Language Models

3.1 Test Case Generation

Traditional software testing requires manual effort in both test case design and implementation. Manually writing test cases in complex systems is particularly time-consuming and prone to omissions. The addition of LLMs opens up new avenues for automated testing. Leveraging their ability to understand natural language requirements and program logic, they can generate test cases based on requirements or code [8]. For example, if a developer enters the question "Does the login function correctly handle incorrect passwords?", the model will generate test code that includes both normal and abnormal paths. Empirical evidence suggests that test cases generated by LLMs match or exceed those written manually in terms of code coverage and defect detection. For example, a study of a Java project found that GPT-4's automatically generated unit tests achieved 85% statement coverage, only about five percentage points less than manually written tests. This demonstrates the practical value of LLMs.

LLMs can also assist CI/CD systems in generating and executing tests after code is submitted, providing timely feedback. However, the test cases generated by this model may contain some redundancy or lack of boundary conditions, and still require human effort to check and improve them. In the future, with the use of technologies such as evolutionary algorithms and symbolic execution, the effectiveness of test case coverage will be improved.

3.2 Defect Detection and Code Review

Defect detection and code review are two key steps in ensuring software quality. Traditional approaches rely on either static analysis tools or manual review, but both methods suffer from insufficient defect coverage and low efficiency. The emergence of LLMs has disrupted this trend. Pre-trained on a large corpus of code, LLMs can learn common programming patterns and anti-patterns and identify potential vulnerabilities [9]. For example, the model can automatically prompt, "This SQL query lacks parameterization, which may lead to an injection risk," and provide a fix. Some open source projects have already embedded LLMs into the GitHub Pull Request process as intelligent review assistants. Empirical evaluations indicate that LLM-based tools can detect a nontrivial proportion of logic bugs and security vulnerabilities—some studies report that LLMs outperform static analyzers in certain contexts. In addition, LLMs can perform semantic verification tasks that go beyond traditional static analysis, such as flagging mismatches between function names and implementations, or detecting inconsistencies not captured by rule-based tools [10]. However, the illusions and error rates of LLMs cannot be ignored. If people blindly follow the model's instructions, false alarms may occur or important issues may be missed. Therefore, we propose a collaborative human-machine review model: the model performs a quick initial screening and produces results, which are then reviewed by engineers and finally decided. This not only improves

work efficiency but also enhances the credibility of the test results. In the future, as model accuracy improves and it is integrated with program analysis tools, intelligent defect detection will become a routine task in software engineering.

3.3 Software Reliability and Verifiability

Software reliability is a critical prerequisite for industrial applications. Yet, the code and testing suggestions produced by LLMs often exhibit ambiguity and lack formal verification, which poses significant risks in safety-critical domains such as healthcare, finance, and aerospace. Therefore, improving the reliability and verifiability of LLM applications has become a common concern in academia and industry. One research approach is the neuro-symbolic approach. LLMs are used to generate candidate code or logic, which are then verified using formal methods such as model checking and symbolic execution [11]. For example, after a model generates a sorting algorithm, formal methods can be used to verify that each input meets stability and correctness standards. This approach combines the creative power of LLMs with the precision of formal methods.

The predictability and consistency of model outputs are also crucial. To address the randomness of content generated by LLMs, researchers have proposed improving stability through prompt engineering and few-shot learning. The industry is also exploring the integration of trusted computing with LLMs to ensure that every step of reasoning in critical systems can be traced and verified.

4. Emerging Challenges in LLM-Driven Software Engineering

4.1 Security and Privacy

The widespread use of LLMs in software engineering inevitably raises security and privacy concerns. First, during the training phase, the training process relies on large-scale code and datasets, which may contain sensitive information such as user accounts, passwords, and internal enterprise logic code. Without effective data desensitization and filtering mechanisms, the model may leak sensitive information during generation. For example, research has found that some LLMs output API keys or internal code from training data when users request them, causing security issues. During inference, developers may enter business requirements, system architecture, or configuration files as prompts into LLMs. If the model is deployed on a third-party cloud, unencrypted data can be misused or stolen. This poses a significant risk to highly sensitive industries such as finance and healthcare. Furthermore, the code generated by the model may introduce new security issues, such as the use of insecure libraries and the lack of input validation. To mitigate these issues, researchers and practitioners are exploring solutions including differential privacy, federated learning, and independent security audits. Yet, these techniques remain limited in scope and cannot completely eliminate risks. In practice, software teams must therefore balance productivity gains with responsible data governance, ensuring that efficiency improvements do not come at the cost of user privacy or system integrity.

4.2 Explainability and Controllability

The “black box” nature of LLMs poses interpretability and controllability challenges when used in software engineering. It is difficult for developers to understand why the model generates certain code or proposes a certain solution, significantly undermining confidence in the model for critical tasks. If the algorithms generated by LLMs involve financial transactions or medical diagnoses, the inability to clearly explain how they make decisions can pose compliance risks and ethical issues for users. To improve interpretability, researchers have incorporated visualization methods into LLMs outputs, highlighting the connections between inputs and outputs to help developers understand the model's inference pathways. Others have proposed “controllable text generation,” allowing developers to apply constraints or rules to guide model output. Restrictions such as “complying with security regulations” or “requiring compatibility with specific libraries” can enhance the controllability of the results.

However, existing methods still have shortcomings. Explanations can only provide "correlation analysis" and fail to delve into the underlying logic of the model's internal representations. Controllability mechanisms can also reduce model creativity due to overly restrictive controls. Therefore, the future trend is "human-machine co-management", which means continuously optimizing the output of LLMs through human supervision and feedback to ensure that it is both fast and controllable in actual applications.

4.3 Intellectual Property and Liability

The use of LLMs in software engineering has also raised disputes over intellectual property and liability. Training data largely comes from open source code repositories or publicly available information on the internet, which may contain code licensed under various open source licenses, such as the GPL or Apache. Models "borrow" these fragments when generating results, but whether this constitutes infringement remains undetermined. For instance, in the case of GitHub Copilot, some users found that the code it generated was highly similar to existing open-source projects, which led to a class-action lawsuit. However, the issue of liability attribution for LLMs remains highly complex. When model-generated code results in security vulnerabilities or financial losses, it is often unclear which party should bear primary responsibility. The current legal frameworks in this area are still underdeveloped: assigning full responsibility to users may hinder the adoption of LLMs, while placing all liability on developers could stifle technological innovation.

Academia and policymakers are exploring solutions. One approach is to establish a "data licensing transparency system" to ensure the traceability of the source of training data; another is to introduce a "responsibility sharing system" whereby algorithm providers and users share the corresponding risks. In addition, international standardization organizations such as ISO and IEEE are also working hard to promote relevant standards in order to provide legal and industry protection for the reasonable use of LLMs in software engineering.

5. Conclusion

In conclusion, this article finds that LLMs are reshaping software engineering with tremendous force. They permeate the entire software lifecycle—from requirements analysis and code generation to testing, debugging, and even ongoing maintenance — providing powerful automation and intelligent support. This has significantly improved productivity and the development experience. However, while LLMs present opportunities, they also raise new challenges, such as security and privacy concerns, difficulties in interpretation, intellectual property rights, and liability issues. These challenges require both technical solutions and the improvement of legal, ethical, and industry standards. In the long term, the connection between LLMs and software engineering will become increasingly close, with human-machine collaboration becoming a predominant model. When achieving quality and sustainable development in software engineering, a balance between efficiency and credibility must be struck. Despite these contributions, this study has several limitations. The analysis primarily relies on existing literature and secondary data, which may not fully reflect the rapidly evolving nature of LLM technologies. In the future, longitudinal and quantitative studies are needed to evaluate the real-world impact of LLMs on software quality, security, and maintainability.

References

[1] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., ... & Wang, H. (2024). Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology, 33(8), 1-79.

[2] Costa, P. M., & Almeida, C. (2024). Innovations in AI and Their Impact on Software Engineering and Beyond. International Journal of Progressive Research in Engineering Management and Science,4(07), 1523-1533.

[3] Shethiya, A. S. (2024). Engineering with Intelligence: How Generative AI and LLMs Are Shaping the Next Era of Software Systems. Spectrum of Research, 4(1).

[4] Liang, J. T., Badea, C., Bird, C., DeLine, R., Ford, D., Forsgren, N., & Zimmermann, T. (2024). Can gpt-4 replicate empirical software engineering research?. Proceedings of the ACM on Software Engineering, 1(FSE), 1330-1353.

[5] Schneider, S., et al. (2025). Analysis of LLMs vs. human experts in requirements engineering. arXiv preprint arXiv:2501.19297.

[6] Pandey, R., Singh, P., Wei, R., & Shankar, S. (2024). Transforming software development: Evaluating the efficiency and challenges of GitHub Copilot in real-world projects. arXiv preprint arXiv:2406.17910.

[7] Zhong, L., Zhang, Y., Xu, H., Wang, Z., & Zhang, M. (2024). Debug like a human: A large language model debugger via verifying runtime execution. arXiv preprint arXiv:2402.16906.

[8] Hasan, S., Anwar, A., & Sarhan, A. (2025). Automatic high-level test case generation using large language models. arXiv preprint arXiv:2503.17998.

[9] 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.

[10] Li, Y., Li, X., Wu, H., Xu, M., Zhang, Y., Cheng, X., Xu, F., & Zhong, S. (2025). Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask. arXiv preprint arXiv:2504.13474.

[11] Lu, Z., Afridi, I., Kang, H. J., Ruchkin, I., & Zheng, X. (2024). Surveying neuro-symbolic approaches for reliable artificial intelligence of things. Journal of Reliable Intelligent Environments, 10(3), 257-279.