Available online at www.rajournals.in

RA JOURNAL OF APPLIED RESEARCH
ISSN: 2394-6709
DOI:10.47191/rajar/v11i9.07
Volume: 11 Issue: 09 September 2025

International Open Access

Impact Factor- 8.553

Page no.- 811-821

Impact of Automated Software Development Using Large Language Models: Capabilities, Limitations, and Future Evolution

MD Raziul Hasan Nayon[1*], S M Shezan Ahmed[2], MD Tariq Uz Zaman[3], MST Ashima Alom Shova[4], Fatin Khan[5]
[1,2,3,4,5]School of Computer Science and Artificial Intelligence, Zhengzhou University, Zhengzhou 450001, Henan, China

ARTICLE INFO

Published Online:
18 September 2025

ABSTRACT

The rapid integration of Large Language Models (LLMs) in the field of software engineering is very much changing the methods of coding, which, at the same time, are also being maintained and optimized. Through this article the journey of the coming of capabilities and restrictions as well as the direction of the future of software development with LLM is monitored. The authors of this article have given a detailed survey of LLM utilization in various stages of the life cycle of development organization, a number of them being code generation, bug detection, automated testing, documentation, and translation of natural language into code, productivity, quality, and accessibility being among the improvements indicated. It is demonstrated through comparative analyses that LLMs' performance is more favorable than that of traditional and rule-based approaches, while the positions of developers, project managers, and executives as stakeholders reveal that they are both excited about the efficiency gains and, on the other hand, concerned about issues of technical reliability, data privacy, and over-reliance on automation. The main problems that LLMs are facing today—like hallucinations, being out-of-date, not having a very large context, and depending too much on prompt engineering—are plainly revealed along with the proposed solutions. Next, we consider progress in the field of multimodal and context-aware systems, autonomous software agents, continuous learning, and human-AI co-creation platforms. LLM-assisted development is likely to change the face of challenges if it is going to be fully materialized, thus bringing about the birth of a new era of collaborative, efficient, and innovative software engineering.

Corresponding Author:
MD Raziul Hasan Nayon

KEYWORDS: Large Language Models (LLMs), automated software development, code generation, bug detection and repair, automated testing, natural language to code translation, prompt engineering, context-aware systems, human-AI collaboration, software engineering automation

1. INTRODUCTION

The software engineering sector is changing in a exponential manner by LLMs which explains the change in concept, creation, management and maintenance of the code [1]. The LLMs-driven automation technique offers a significant increase in productivity along with the potential for higher code quality, shorter delivery time, and the opening of coding to a larger number of people by reducing the technical barriers [1,2]. The expanding scope and complexity of projects highlight the need for smart, context-aware, and flexible tools which has led to the fast adoption of LLMs in different parts of software development.

LLMs are at the heart of empowering the next phase in software engineering by facilitating tasks that were very labor-intensive earlier [3]. The range of skills they have incorporates code generation, auto-completion, defect detection, and documentation, as well as translating natural language to code [4]. As a result, software engineers are changing their roles from merely writing code to supervising, directing, and checking AI-generated work; thus, not only this but also the new forms of human-AI collaboration are becoming possible [1].

This research is designed to methodically identify the influence on software automation of LLMs. The research aims to go beyond the present capabilities, uncover limitations and difficulties, offer comparisons with traditional and rule-based systems, survey stakeholders' positions, furnish examples of the real world, and speculate on the future of LLMs in software engineering. The paper discusses the technical, organizational, and societal aspects of LLM-fueled automation in software development. This article presents a deep literature review, empirical feature analysis, problem identification, comparative evaluation, discussion of

lifecycle impact, stakeholders' analysis, industry case studies, and the outlook on the development of LLMs in engineering practice.

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\top}}{\sqrt{d_k}}\right)\boldsymbol{V}$$

LLM Attention Mechanism (Transformer architecture)

2. LITERATURE REVIEW

Research on LLMs in software engineering has grown exponentially, with surveys and systematic literature reviews revealing their game-changing impact throughout the software development lifecycle [5,6]. Studies are enumerating remarkable progress in code generation and automated programming [7], the incorporation of LLMs into developer tooling and IDEs [2], and the facilitation of testing, documentation, and maintenance processes [4,8].The literature also highlights challenges, such as model hallucinations, context awareness deficiencies, data privacy implications, and new forms of workflow integration [9,10]. Comparative research shows LLMs, when fine-tuned and combined with optimized prompts, frequently outperform traditional rule-based or search-based coding assistants, though concerns about output reliability and context dependence persist [11]. The significance of domain-specific benchmarks, real-world codebases, and interdisciplinary perspectives in evaluating LLMs' practical effectiveness is increasingly recognized in recent research [12,13].

The complexity of modern software systems has also led to the increasing use of software models and Model-Driven Development (MDD) [14,15]. MDD aims to enhance quality and speed by enabling the development of more accurate models through tools like the Object Constraint Language (OCL) [16]. Model transformations and compositions are central to MDD, facilitating the development of complex systems and their automated derivation [14]. However, there are still challenges regarding the validity of such transformations, highlighting the need for safe composition and decomposition of models [14]. Textual representations of software models can also be compared and merged, which aids in optimizing teamwork through version control systems [15].

3. CAPABILITIES OF LLMS IN SOFTWARE DEVELOPMENT

3.1 Code Generation and Auto-Completion

LLMs have fundamentally reshaped code generation and auto-completion processes by delivering high-quality, contextually relevant code suggestions based on natural language prompts or preceding code fragments [17]. Advanced models such as GPT-4, Llama 3.1, and specialized systems like ProgAI and DevAssistLlama have proven adept at synthesizing code in multiple programming languages, providing both token-level completions and generating entire functional blocks that closely match the user's intent[17,18]. Recent benchmarks show that, although performance dips on more intricate, real-world tasks, LLMs have achieved improvements of 18.1% to 25% in code generation accuracy over traditional tools, with specialized frameworks like ProgAI even surpassing commercial products such as GitHub Copilot in repo-level coding tasks[17,19]. To address challenges like hallucinations—where models generate non-existent APIs or incorrect code—strategies including iterative context augmentation (e.g., De-Hallucinator) and graph-based context retrieval (e.g., GraphCoder) have effectively reduced erroneous output and improved the integration of repository-specific knowledge into code completions[19].

3.2 Bug Detection and Fixing

LLMs have accelerated bug detection and fixing by leveraging their understanding of both code and natural language descriptions of bugs, enabling automated identification, localization, and patch generation for software errors[20]. Knowledge-enhanced LLMs use external bug knowledge graphs, mining historical bug data and fix patterns from platforms like GitHub and Stack Overflow to suggest accurate, explainable patches, achieving correct fixes in 28.52% of evaluated cases, significantly outperforming conventional approaches [20]. Moreover, advanced agent-based frameworks such as PATCH and STEAM simulate the multi-stage, collaborative debugging process by decomposing the bug-fixing lifecycle into sequential phases—such as reporting, diagnosis, patch generation, and verification—performed interactively by LLMs [21]. Techniques like Tree of Thoughts prompting encourage exploration of multiple hypotheses before code modification, enabling resolution of complex bugs previously unsolved by earlier methods[22]. While LLMs can outperform deep learning-based tools and standard automated program repair technologies, their efficacy depends on the richness of provided context and model capabilities for collaborative reasoning[20].

3.3 Automated Testing and Test Case Generation

The automation of software testing is a major achievement enabled by LLMs, which address the traditionally labor- and time-intensive nature of test case development[23]. LLMs like GPT and Mistral excel in generating unit tests, achieving high coverage for a wide array of programming constructs and providing more natural, human-readable test cases compared to existing automatic tools[23]. Advanced prompt engineering techniques and retrieval-augmented frameworks further enhance this capability, enabling models to generate tests with increased correctness, understandability, and breadth of coverage, often matching or exceeding state-of-the-art tools like EvoSuite and traditional rule-based generators[24]. Industrial deployments demonstrate their value, with LLM-powered agents automating entire test generation and execution pipelines, as evidenced in cost-

effective UI automation platforms that realize up to 90% automation rates at lower costs compared to other methods[25]. Nonetheless, improvements are still needed to reduce issues such as test smells and improve logical path coverage in complex software[26].

3.4 Code Summarization and Documentation

LLMs significantly enhance code summarization and documentation generation by programmatically producing natural language descriptions for code at varying granularities, from individual statements to repository-level overviews[27,28]. They facilitate improved code comprehension and maintenance by automatically generating explanatory comments, API references, and detailed summaries, thereby reducing the manual documentation burden for developers[29,30]. Approaches leveraging hierarchical summarization and context-rich retrieval (e.g., RAG and prompt learning frameworks) have further advanced performance, enabling models to contextualize summaries in terms of domain or business logic[27,31]. LLMs such as Codex, RoBERTa, and instruction-fine-tuned GPT variants have surpassed legacy approaches in key metrics, producing more accurate and readable documentation across multiple languages and domains[32,33]. Specialized evaluation of statement-level and function-level summarization reveals that LLMs rival or outperform pre-trained models like CodeT5, and empirical studies with professional developers highlight the increased naturalness and usability of LLM-generated documentation[34,35].

3.6 Natural Language to Code Translation

Empowered by neural machine translation advances, LLMs can interpret high-level human instructions and generate corresponding source code, accelerating prototyping and making software development more accessible to non-developers [36] . Major models like GPT-4, Llama 3.1, and domain-adapted frameworks achieve strong results in mapping user requirements to code across numerous general-purpose and domain-specific languages, including support for complex code edits in response to bug fix reports [37]. Empirical studies show top-1 correctness rates exceeding 20% for challenging bug-fix tasks and that models can even outperform human translators in certain formal verification contexts, such as translating requirements into formal Computation Tree Logic (CTL) specifications [38].

4. LIMITATIONS AND CHALLENGES

4.1 Technical Limitations: Hallucinations and Outdated Knowledge

Despite their sophisticated architectures and impressive performance, LLMs are inherently susceptible to generating hallucinations—outputs that appear plausible but are factually incorrect, misleading, or entirely fabricated[39]. In software engineering settings, hallucinations might manifest as non-existent APIs, syntactically valid but incorrect code, or unsubstantiated claims in code documentation, posing severe reliability and security risks[40][41]. Empirical studies have revealed high hallucination rates in critical domains, with legal and technical LLMs hallucinating over 50% of the time in benchmarks[39].

Several mitigation strategies exist, including retrieval-augmented generation (RAG), systematic prompt engineering, structured output constraints, and knowledge graph integration, which can reduce but not fully eliminate the problem[41,42]. Human oversight remains essential in reviewing, verifying, and correcting LLM-generated code and documentation, underscoring the ongoing need for manual quality assurance[42].

Outdated knowledge constitutes another critical technical limitation[43,44]. Most LLMs are trained on large text corpora with a fixed cutoff date, resulting in responses that may reference deprecated APIs, obsolete security practices, or superseded frameworks[43]. This limitation is particularly severe in the fast-moving field of software engineering, where tools and best practices evolve rapidly[44]. Methods such as continual learning, dynamic knowledge editing, and retrieval of up-to-date external information have been proposed, but current models still frequently provide outdated or incorrect recommendations, especially for reasoning tasks[43,44].

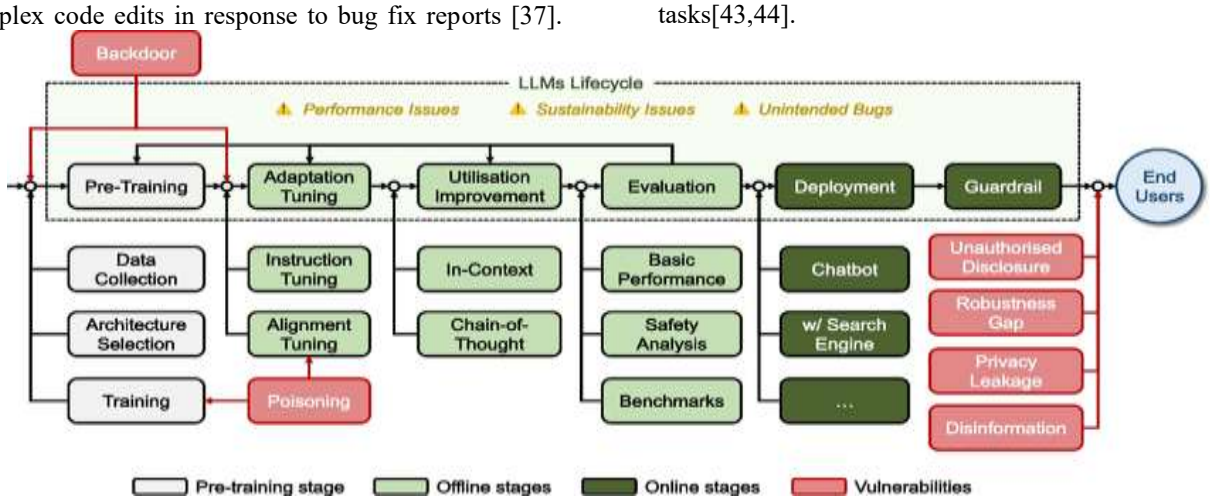

Figure 1. LLM's lifecycle.

4.2 Context and State Awareness Challenges

Maintaining context and handling long-term state are pronounced hurdles for LLMs in complex software development tasks[45]. Standard context windows restrict the amount of code or documentation an LLM can process at once, resulting in context loss or inaccuracies when working with large codebases or multi-turn coding sessions[45,46]. As the required context length grows, models exhibit brittle behavior and a decline in instruction fidelity, leading to degraded code generation quality or logical inconsistencies[47].

In vulnerability detection and debugging, limited context comprehension hampers the identification of intricate security flaws, while in guided code generation, LLMs can falter in compositional reasoning across lengthy or fragmented code files[46]. Techniques such as prompt memorization, multi-agent decomposition, extended context windows, and memory-augmented architectures like FastMem have shown promise, but fundamental context-awareness issues still inhibit stable and reliable performance in end-to-end software development workflows[47].

4.3 Data Privacy and Security Concerns

Data privacy and security are paramount concerns for LLM-driven software engineering, as these models are often trained on vast, uncurated datasets that may inadvertently include proprietary or sensitive code, confidential user information, or personal data[34,48]. This introduces significant risks of unintentional privacy leakage, both during training—when sensitive data may be memorized—and at inference, where LLMs might accidentally or maliciously output private or regulated content when prompted[48].

Risks are exacerbated in enterprise and multi-user settings, where a single model may see source inputs from multiple organizations, complicating access control and privacy boundaries[49]. Malicious attacks such as membership inference or model inversion have been demonstrated, highlighting vulnerabilities in even privacy-protected or federated learning setups[27]. To tackle these issues, researchers advocate differential privacy during training, federated and decentralized learning paradigms, cryptographically secure computation, and privacy-preserving association editing—methods that can reduce but rarely completely remove leakage risk [48]. Organizations must maintain vigilant procedural oversight, establish policy frameworks, and develop clear governance strategies to ensure regulatory compliance and safeguard IP and user privacy in LLM-assisted workflows[34].

4.4 Dependence on Prompt Engineering

LLM performance is highly sensitive to input prompt formulation, making prompt engineering—a new discipline focused on optimizing prompt design—central to realizing the potential of automated code generation[28]. The ability of LLMs to handle complex software engineering tasks, generate robust code, or answer domain-specific questions is primarily governed by specificity, clarity, and structure of prompts[32]. This dependency creates several bottlenecks:

The trial-and-error nature of prompt development can be inefficient, ad hoc, and costly in both time and computational resources [28,29].

- Suboptimal prompts may trigger hallucinations, generate insecure or buggy code, or fail to satisfy detailed requirements [31].
- As LLMs evolve, earlier prompt engineering methods may lose efficacy, and advanced models may even underperform with overly complex prompts, necessitating ongoing research and adaptation [31].

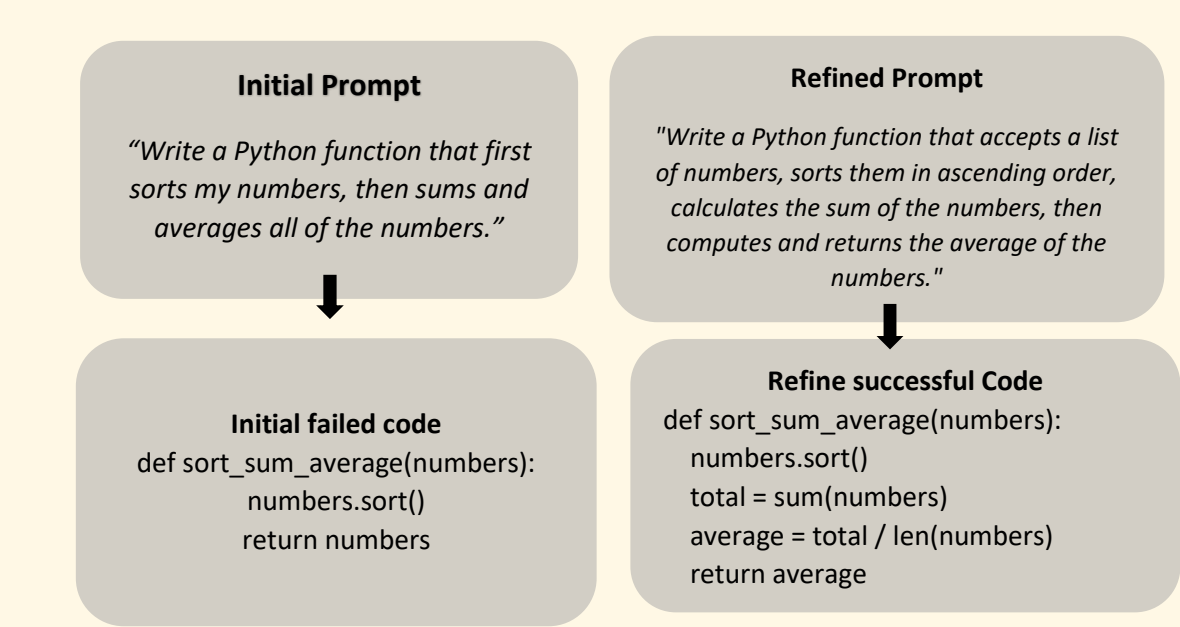

Figure 2. Effects of prompt engineering

Although automation techniques such as search-based prompt optimization, role-prompting, chain-of-thought, and retrieval-augmented strategies improve LLM outputs, the field still lacks standardized methodologies, leading to what some researchers have termed a "promptware crisis" in software engineering [28,29].

5. COMPARATIVE ANALYSIS

5.1 Traditional Development vs. LLM-Assisted Development

The transition from traditional to LLM-assisted software development introduces significant shifts in speed, efficiency, and accessibility. While traditional development requires explicit programming knowledge and time-consuming debugging, LLMs like Copilot and ChatGPT-4 offer real-time code generation, significantly accelerating tasks [50]. demonstrated that developers using GitHub Copilot completed tasks 55.8% faster than those using manual coding approaches.

Table 1. Traditional Development vs. LLM-Assisted Development

Metric	Traditional Dev	LLM-Assisted Dev (Copilot)
Task Completion Time	Baseline	55.8% faster
Developer Productivity	Normal pace	Significantly improved
Time to Fix Bugs	Manual debugging	AI-suggested fixes

5.2 Human vs. LLM-Generated Code: Quality & Productivity

compared Claude, GPT-4, Copilot, and Gemini against human performance on 200 programming questions. Claude and GPT-4 outperformed both students and other LLMs with 83% and 81.7% accuracy, respectively, while Gemini achieved only 53.6% [51].

Table 2. Human vs. LLM-Generated Code: Quality & Productivity

Model	Accuracy (%)	Outperformed Humans?
Claude 3.5	83.0	Yes
GPT-4	81.7	Yes
Copilot	59.5	No
Gemini	53.6	No
Students	67.2	Baseline

5.3 Performance Comparison of Major LLMs

Recent evaluations highlight the technical superiority of GPT-4o and Claude 3.5 over alternative models. GPT-4o achieved a balanced accuracy of 96.3% on structured data tasks, while Claude Sonnet 3.5 led in whole-scan tasks for real-world document parsing[52,53].

Table 3. Performance Comparison of Major LLMs

Model	Accuracy (code/tech tasks)	Notes
Claude 3.5	83.0%	Best for structured technical domains
GPT-4o	81.7%–96.9% (task-dependent)	Highly consistent & reproducible
Copilot	~59.5%	Best for integration in IDEs
Gemini	~53.6%	Lower coding precision

6 STAKEHOLDER PERSPECTIVES

The use of Large Language Models (LLMs) in software development has greatly complicated the perceptual landscape among numerous stakeholders. The parties involved are characterized by mixed feelings of deep faith in the revolutionary potential of LLMs and anxiety over their shortcomings and danger. The research cited in [35, 54] provides examples of such situations. An awareness of these various outlooks is key to successfully steering the implementation and subsequent growth of LLMs in software engineering.

6.1 Software Developers

Software developers, as direct users of LLMs, generally perceive these tools as highly beneficial for enhancing productivity and streamlining routine tasks [30,35,54]They report that LLMs are particularly helpful for generating foundational code structures, assisting with C++ specific syntax, debugging errors, and improving code quality [54]. LLMs also serve as a "pair-programmer" for developers, facilitating quicker understanding and debugging of error messages [54]. Studies indicate that LLMs can significantly reduce task completion time, with some reporting over a 50% reduction [30,33]. Beyond direct coding, LLMs assist developers in learning new programming languages and concepts, aiding in personal and professional development [35]. Despite these advantages, developers face several challenges [54]. They express concerns that over-reliance on LLMs might deprive them of gaining fundamental coding skills or negatively impact the software engineering job market [54]. Technical inaccuracies, often referred to as "hallucinations," and the generation of misleading or incorrect content necessitate careful manual review before integrating LLM outputs into production environments [33,35]. Prompt engineering, the process of crafting effective queries for LLMs, adds cognitive load and requires a specific skill set, which can be a barrier for some users [30]. Interestingly, more skilled developers are often more inclined to use AI code generation tools, suggesting that effective utilization requires a foundational understanding of coding to vet and manipulate the AI-generated code [54].

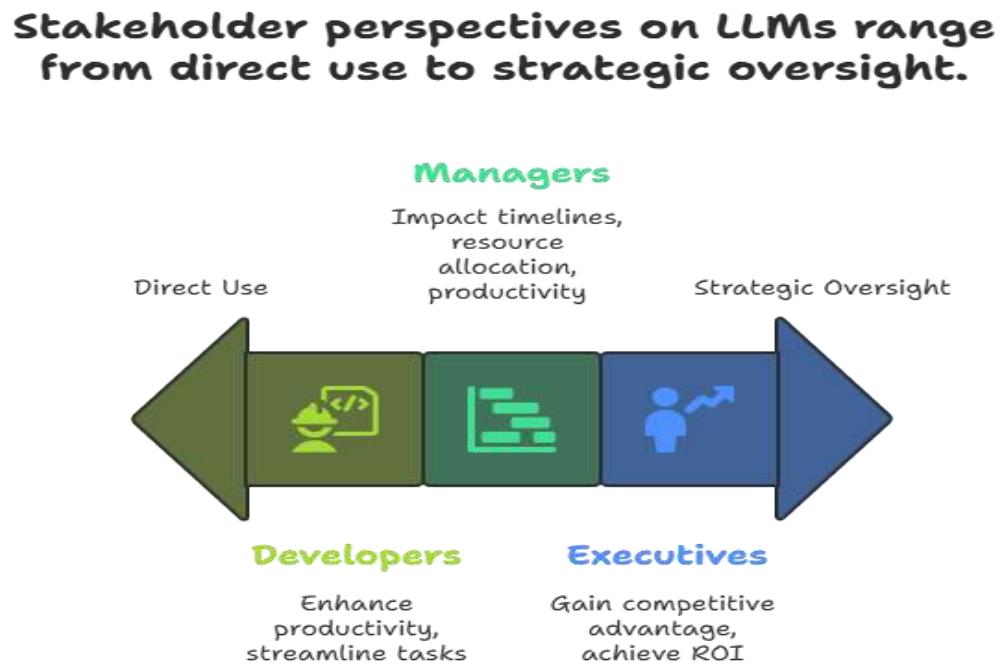

Figure 3. Stakeholder perspectives on UMs range from direct use to strategic oversight.

6.2 Project Managers

Project managers recognize LLMs as powerful tools that can significantly impact project timelines, resource allocation, and overall productivity [30,55]. They foresee accelerated development cycles due to LLMs automating repetitive coding tasks, debugging, and potentially optimizing code, which can lead to faster time-to-market [55]. LLMs can assist in various project management aspects by processing large amounts of project data, aiding in requirements analysis, and providing insights for resource scheduling [55]. This capability helps in more precise task assignments and optimizing team utilization [55]. However, project managers are also wary of the risks associated with over-reliance on AI-generated outputs, which may introduce inaccuracies or biases from the training data [55,56]. They emphasize the continued need for human oversight to validate AI-generated code and decisions [55]. The incorporation of LLMs to work with already existing project workflows generally entails changes in management styles and it may encounter resistance from the organization [55]. Project managers' concerns about data privacy, intellectual property rights, and the ethical treatment of AI tools are very important and thus, they have to be given due care and clear instructions while implementing this [55].

6.3 Business Leaders and Executives

Business leaders and executives consider LLMs to be a strategic factor that secures their position in the market and guarantees a significant return of investment (ROI) in software development [57]. They understand the impact that LLMs have on automating the routine operations, speeding up the work processes, and improving the product characteristics, thereby enabling operational efficiency to be improved and innovation to be faster [58]. The feature of LLMs to cut down on the time used for non-creative work allows developers to direct their efforts towards solving more complicated problems and creating the activities that add the most value. Early adoption is regarded as being critical for retaining a competitive edge in the technology sector that is changing fast [3]. Nevertheless, this positive outlook is moderated by the acknowledgment of significant problems [57]. Executives reveal that they are worried about the computational costs that come along with the activities of both training and inferring LLMs, which are stated to be much higher than the costs of traditional software systems [3]. Besides that, they are aiming at surmounting the resistance that the organizations hold against the new technologies and managing the problem of incorporating LLMs into the existing IT infrastructures if they are to succeed [58]. Furthermore, business leaders are clearly at the forefront of ethics issues related to the biases in the training data and the lack of transparency when it comes to the LLM decision-making that they see as the main factors that cannot be ignored if fair and inclusive software solutions are to be realized. At the same time, legal and reputational risks also come into the equation [3].

7. FUTURE EVOLUTION OF LLMS IN SOFTWARE ENGINEERING

The trajectory of large language models (LLMs) in software engineering suggests a profound and ongoing transformation—both in technological capability and in the roles of human developers. While current LLMs have already demonstrated remarkable aptitude in tasks such as code generation, debugging, and documentation, their future evolution is poised to address existing limitations and unlock new paradigms in automated software development.

7.1 Multimodal and Context-Aware Systems

The path of LLMs in software engineering indicates a far reaching and continuous change in both technological features and the roles of people who do the jobs. Current LLMs are still good at code generating, error detection, and documenting but their future development is expected to solve the problems they have and bring in new automated software development scenarios.

7.2 Autonomous Software Agents

As LLMs evolve into agent-like structures, we are expecting the emergence of independent software engineering agents that can carry out full-stack jobs with very little help. Such agents will handle end-to-end pipelines—including tasks like understanding the vaguest customer needs to actually deploying systems—by employing memory-augmented architectures, self-revision loops, and multi-agent collaboration techniques. OpenAI's AutoGPT, Google's Gemini agents, and DeepSeekCoder are examples of this trend in its infancy.

Figure 4. The future of software development: LLM Developers.

7.3 Continual Learning and Real-Time Adaptation

Present-day LLMs have intrinsic limitations such as knowledge cutoffs and they are in need of retraining from time to time. However, upcoming models will feature **continual learning** capabilities, allowing them to learn new things on the go and access new frameworks,

languages, and APIs without having to go through complete retraining. Such developments will not only reduce the technical debt but also facilitate the synchronization of the models with the ever-changing software ecosystem. Strategies like parameter-efficient fine-tuning (PEFT) and retrieval-augmented

7.4 Human-AI Co-Creation Platforms

Instead of replacing programmers, LLMs are anticipated to be co-creative partners. Next-generation Integrated Development Environments (IDEs) will seamlessly integrate LLMs as smart, conversational buddies who adapt to developer styles, suggest changes actively, and give understandable explanations for the created code. The human-in-the-loop model will transform to a **symbiotic partnership**, in which both machine intelligence and human skills are continuously used to improve results.

8 CONCLUSION

The incorporation of large language models (LLMs) into programming indicates a major change in the field of software engineering. LLMs have proven to be very effective in increasing productivity, improving the quality of code, shortening delivery times, and making coding more accessible by lowering the technical barriers. Their usages cover different phases of software development, starting with code generation and auto-completion and ending with bug detection, automated testing, code summarization, and also natural language to code translation. The developments in LLMs have resulted in big gains in efficiency and accessibility which give developers the opportunity to concentrate more on complex, value-adding tasks instead of routine coding jobs. On the other hand, the use of LLMs in software development also has some problems like technical limitations, hallucinations, and outdated knowledge, context and state awareness difficulties, data privacy and security issues, and a dependence on prompt engineering. However, the future of LLMs in software engineering is very bright. When LLMs go further, we will be able to see more multimodal and context-aware systems, autonomous software agents, continual learning and real-time adaptation, and also human-AI co-creation platforms. These coming facilities will make LLMs even more powerful and help them to get rid of some of their current weaknesses. The smooth integration of LLMs into programming will still need a cautious dealing with the complex environment of stakeholder perspectives, including those of software developers, project managers, and business leaders. By solving the problems and making full use of the potentials of LLMs, the software engineering industry will be able to look ahead to a future when human and AI collaboration will be the main driving force behind innovation and efficiency in software development.

REFERENCES

1. Terragni V, Vella A, Roop P, Blincoe K. The Future of AI-Driven Software Engineering. ACM Transactions on Software Engineering and Methodology 2025;34:1–20. https://doi.org/10.1145/3715003.
2. Peng X. Software development in the age of intelligence: embracing large language models with the right approach. Frontiers of Information Technology & Electronic Engineering 2023;24:1513–9. https://doi.org/10.1631/FITEE.2300537.
3. Haque MdA. LLMs: A game-changer for software engineers? BenchCouncil Transactions on Benchmarks, Standards and Evaluations 2025;5:100204. https://doi.org/10.1016/j.tbench.2025.100204.
4. Husein RA, Aburajouh H, Catal C. Large language models for code completion: A systematic literature review. Comput Stand Interfaces 2025;92:103917. https://doi.org/10.1016/j.csi.2024.103917.
5. Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, et al. Large Language Models for Software Engineering: Survey and Open Problems. 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), IEEE; 2023, p. 31–53.https://doi.org/10.1109/ICSE-FoSE59343.2023.00008.
6. Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, et al. Large Language Models for Software Engineering: A Systematic Literature Review. ACM Transactions on Software Engineering and Methodology 2024;33:1–79. https://doi.org/10.1145/3695988.
7. Wang J, Chen Y. A Review on Code Generation with LLMs: Application and Evaluation. 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI), IEEE; 2023, p. 284–9. https://doi.org/10.1109/MedAI59581.2023.00044.
8. Yu S, Fang C, Ling Y, Wu C, Chen Z. LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities. 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), IEEE; 2023, p. 206–17. https://doi.org/10.1109/QRS60937.2023.00029.
9. Rasnayaka S, Wang G, Shariffdeen R, Iyer GN. An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project. Proceedings of the 1st International Workshop on Large Language Models for Code, New York, NY, USA: ACM; 2024, p. 111–8. https://doi.org/10.1145/3643795.3648379.
10. Nunes H, Figueiredo E, Rocha L, Nadi S, Ferreira F, Esteves G. Evaluating the Effectiveness of LLMs in

Fixing Maintainability Issues in Real-World Projects. 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE; 2025, p. 669–80. https://doi.org/10.1109/SANER64311.2025.00069.

11. Jahić J, Sami A. State of Practice: LLMs in Software Engineering and Software Architecture. 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), IEEE; 2024, p. 311–8.https://doi.org/10.1109/ICSA-C63560.2024.00059.
12. Zhang Q, Fang C, Xie Y, Zhang Y, Yang Y, Sun W, et al. A Survey on Large Language Models for Software Engineering 2024.
13. Wang R, Guo J, Gao C, Fan G, Chong CY, Xia X. Can LLMs Replace Human Evaluators? An Empirical Study of LLM-as-a-Judge in Software Engineering 2025. https://doi.org/10.1145/3728963.
14. Meedeniya D, Perera I, Bowles J. Transformation and composition of software design models for Model Driven Development. 2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS), IEEE; 2015, p. 31–6. https://doi.org/10.1109/ICIINFS.2015.7398981.
15. Somogyi FA. Merging Textual Representations of Software Models. The publications of the MultiScience - XXX. MicroCAD International Scientific Conference, University of Miskolc; 2016. https://doi.org/10.26649/musci.2016.060.
16. Hrycyk JC, Soares IW, Wiedermann Agner LT. Definition of Object Constraint Language Rules in Models to Support the Development of Android Applications. Journal of Computer Science 2018;14:253–9. https://doi.org/10.3844/jcssp.2018.253.259.
17. A AA, D H, Nivedha MrsS. ProgAI: Enhancing Code Generation with LLMs For Real World Challenges. INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT 2024;08:1–11. https://doi.org/10.55041/IJSREM36242.
18. Banerjee S, Dutta A, Layek S, Sahoo A, Joyce SC, Hazra R. Redefining Developer Assistance: Through Large Language Models in Software Ecosystem 2024.
19. Eghbali A, Pradel M.De-Hallucinator: Mitigating LLM Hallucinations in Code Generation Tasks via Iterative Grounding 2024.
20. Bo L, He Y, Sun X, Ji W, Wu X. A Software Bug Fixing Approach Based on Knowledge-Enhanced Large Language Models. 2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS), IEEE; 2024, p. 169–79. https://doi.org/10.1109/QRS62785.2024.00026.
21. Zhang Y, Jin Z, Xing Y, Li G. STEAM: Simulating the InTeractive BEhavior of ProgrAMmers for Automatic Bug Fixing 2023. https://doi.org/10.1145/3718739.
22. Weng G, Andrzejak A. Automatic Bug Fixing via Deliberate Problem Solving with Large Language Models. 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE; 2023, p. 34–6. https://doi.org/10.1109/ISSREW60843.2023.00040.
23. Ouedraogo WC, Kabore K, Tian H, Song Y, Koyuncu A, Klein J, et al. LLMs and Prompting for Unit Test Generation: A Large-Scale Evaluation. Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA: ACM; 2024, p. 2464–5. https://doi.org/10.1145/3691620.3695330.
24. Deng Y, Chen R, Xiao C, Yang Z, Luo Y, Zhao J, et al. LLM - TG: Towards Automated Test Case Generation for Processors Using Large Language Models. 2024 IEEE 42nd International Conference on Computer Design (ICCD), IEEE; 2024, p. 389–96. https://doi.org/10.1109/ICCD63220.2024.00066.
25. Feng S, Lu H, Jiang J, Xiong T, Huang L, Liang Y, et al. Enabling Cost-Effective UI Automation Testing with Retrieval-Based LLMs: A Case Study in WeChat 2024.
26. Wang W, Yang C, Wang Z, Huang Y, Chu Z, Song D, et al. TESTEVAL: Benchmarking Large Language Models for Test Case Generation 2025.
27. Vu MN, Nguyen T, Jeter TR, Thai MT. Analysis of Privacy Leakage in Federated Large Language Models 2024.
28. Chen Z, Wang C, Sun W, Yang G, Liu X, Zhang JM, et al. Promptware Engineering: Software Engineering for LLM Prompt Development 2025.
29. Taherkhani H, Sepindband M, Pham HV, Wang S, Hemmati H. EPiC: Cost-effective Search-based Prompt Engineering of LLMs for Code Generation 2024.
30. Weber T, Brandmaier M, Schmidt A, Mayer S. Significant Productivity Gains through Programming with Large Language Models. Proc ACM Hum Comput Interact 2024;8:1–29. https://doi.org/10.1145/3661145.
31. Wang G, Sun Z, Gong Z, Ye S, Chen Y, Zhao Y, et al. Do Advanced Language Models Eliminate the Need for Prompt Engineering in Software Engineering? 2024.
32. Son M, Won Y-J, Lee S. Optimizing Large Language Models: A Deep Dive into Effective Prompt Engineering Techniques. Applied Sciences

2025;15:1430. https://doi.org/10.3390/app15031430.
33. Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, et al. Large Language Models for Software Engineering: Survey and Open Problems. 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), IEEE; 2023, p. 31–53.https://doi.org/10.1109/ICSE-FoSE59343.2023.00008.
34. Yang J. Large language models privacy and security. Applied and Computational Engineering 2024;76:177–88. https://doi.org/10.54254/2755-2721/76/20240584.
35. Tabarsi B, Reichert H, Limke A, Kuttal S, Barnes T. LLMs' Reshaping of People, Processes, Products, and Society in Software Development: A Comprehensive Exploration with Early Adopters 2025.
36. Ehsan U, Harrison B, Chan L, Riedl MO. Rationalization. Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society, New York, NY, USA: ACM; 2018, p. 81–7. https://doi.org/10.1145/3278721.3278736.
37. Fakhoury S, Chakraborty S, Musuvathi M, Lahiri SK. Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions 2023.
38. Zrelli R, Misson HA, Ben Attia M, de Magalhaes FG, Shabah A, Nicolescu G. Advancing Formal Verification: Fine-Tuning LLMs for Translating Natural Language Requirements to CTL Specifications. 2024 International Workshop on Rapid System Prototyping (RSP), IEEE; 2024, p. 21–7. https://doi.org/10.1109/RSP64122.2024.10870993.
39. Dahl M, Magesh V, Suzgun M, Ho DE. Large Legal Fictions: Profiling Legal Hallucinations in Large Language Models 2024. https://doi.org/10.1093/jla/laae003.
40. Tian Y, Yan W, Yang Q, Zhao X, Chen Q, Wang W, et al. CodeHalu: Investigating Code Hallucinations in LLMs via Execution-based Verification 2025.
41. Sun W, Li B, Zhang GL, Yin X, Zhuo C, Schlichtmann U. Classification-Based Automatic HDL Code Generation Using LLMs 2024.
42. Bhattacharya R. Strategies to mitigate hallucinations in large language models. Applied Marketing Analytics: The Peer-Reviewed Journal 2024;10:62. https://doi.org/10.69554/NXXB8234.
43. Tang W, Cao Y, Deng Y, Ying J, Wang B, Yang Y, et al. EvoWiki: Evaluating LLMs on Evolving Knowledge 2024.
44. Sun Z, Liu Y, Wang J, Meng F, Xu J, Chen Y, et al. Outdated Issue Aware Decoding for Reasoning Questions on Edited Knowledge 2024.
45. Gupta L, Sharma S, Zhao Y. Systematic Evaluation of Long-Context LLMs on Financial Concepts. Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track, Stroudsburg, PA, USA: Association for Computational Linguistics; 2024, p. 1163–75. https://doi.org/10.18653/v1/2024.emnlp-industry.88.
46. Almorsi A, Ahmed M, Gomaa W. Guided Code Generation with LLMs: A Multi-Agent Framework for Complex Code Tasks 2025.
47. Zhang Y, Li J, Liu P. Extending LLMs' Context Window with 100 Samples 2024.
48. Venditti D, Ruzzetti ES, Xompero GA, Giannone C, Favalli A, Romagnoli R, et al. Enhancing Data Privacy in Large Language Models through Private Association Editing 2024.
49. Zhan X, Seymour W, Such J. Beyond Individual Concerns: Multi-user Privacy in Large Language Models. ACM Conversational User Interfaces 2024, New York, NY, USA: ACM; 2024, p. 1–6. https://doi.org/10.1145/3640794.3665883.
50. Peng S, Kalliamvakou E, Cihon P, Demirer M. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot 2023.
51. Mavrych V, Yaqinuddin A, Bolgova O. Claude, ChatGPT, Copilot, and Gemini performance versus students in different topics of neuroscience. Adv Physiol Educ 2025;49:430–7. https://doi.org/10.1152/advan.00093.2024.
52. Wulcan JM, Jacques KL, Lee MA, Kovacs SL, Dausend N, Prince LE, et al. Classification performance and reproducibility of GPT-4 omni for information extraction from veterinary electronic health records. Front Vet Sci 2025;11. https://doi.org/10.3389/fvets.2024.1490030.
53. Kim S, Baudru J, Ryckbosch W, Bersini H, Ginis V. Early evidence of how LLMs outperform traditional systems on OCR/HTR tasks for historical records 2025.
54. Rasnayaka S, Wang G, Shariffdeen R, Iyer GN. An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project 2024.
55. Karnouskos S. The Relevance of Large Language Models for Project Management. IEEE Open Journal of the Industrial Electronics Society 2024;5:758–68. https://doi.org/10.1109/OJIES.2024.3412222.
56. Khomh F. Harnessing Predictive Modeling and Software Analytics in the Age of LLM-Powered Software Development (Invited Talk). Proceedings

of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering, New York, NY, USA: ACM; 2023, p. 1–1. https://doi.org/10.1145/3617555.3634736.

57. Zhang L-J, Chen H, He S, Li C, Chen J, Zhang H, et al. COSIS: An AI-Enabled Digital Transformation Framework Integrating Large Language Models and Key Performance Indicators, 2025, p. 74–99. https://doi.org/10.1007/978-3-031-77000-5_6.
58. Pashchenko DS. Early Formalization of AI-tools Usage in Software Engineering in Europe: Study of 2023. International Journal of Information Technology and Computer Science 2023;15:29–36. https://doi.org/10.5815/ijitcs.2023.06.03.