# The Impact of Modern AI on Software Development: A Systematic Literature Review

Meryem Bensaid[1]*, Marouane Achbari[2], Younesse Ouahbi[3], Soumia Ziti[4]
IPSS-Laboratory of Intelligent Processing and Security of Systems, Mohammed V University, Rabat, Morocco[1,2,3,4]
SMSD, Moroccan Society of Digital Health, Rabat, Morocco[4]
LIAS-Artificial Intelligence and Systems Laboratory, Hassan II University, Casablanca, Morocco[4]

***Abstract*—As large language models, and agentic AI systems are increasingly being integrated into software engineering, an expanding amount of empirical evidence surrounding these technologies has emerged. This systematic literature review examines the impact of modern AI techniques and tools in software development lifecycle phases and related activities, covering studies published between 2023 and 2025 and resulting in a corpus of 62 primary studies, investigating the role of large language models, AI agents and agentic AI workflows across lifecycle phases. The synthesis is guided by three research questions that address the entire software development cycle, reported impacts on development practices, outcomes, and constraints. This review fills a gap in the synthesis of modern AI applications and their impacts. It concludes that modern AI in software engineering is progressively evolving from a generation tool into a reasoning and coordination infrastructure layer, with ongoing efforts targeting the mitigation of identified limitations and the advancement of trustworthy agentic AI capabilities for dependable software engineering practices.**

***Keywords*—Large language models; generative AI; agentic AI; software development lifecycle; systematic literature review**

## I. Introduction

Software development has changed significantly since large language models were released in late 2022. Targeted and customized automation is no longer the exclusive application of AI in this field. Every level of software development today benefits from the use of large language models, agents, and agent-based AI systems. These technologies now assist developers in defining requirements, making architectural decisions, writing code, fixing bugs, reviewing pull requests, detecting security risks, and managing production occurrences. As a result, AI is currently being used in majority of development workflows, tools, and infrastructures to help with software system design, implementation, testing, deployment, and maintenance.

Simultaneously, the quantity of empirical research on AI in software engineering has increased significantly. The majority of these studies concentrate on certain tools, jobs, or single lifecycle stages like program repair or code generation. They generate separate understandings even though they offer valuable insights. However, both researchers and practitioners still lack clarity on where modern AI works best, the risks it presents (especially with more autonomous agentic systems), and how its use should be governed. The lack of an integrated perspective makes it difficult to draw meaningful conclusions about the overall impact of modern AI on software development and the new organizational and technological challenges that come with it.

In order to address this gap, this study provides a comprehensive analysis of peer-reviewed primary research on the application of modern AI in software development published between 2023 and 2025 . The review adheres to the PRISMA 2020 guidelines and is organized around a 4-phase SDLC framework: Inception, Construction, Quality Assurance, and Operations. In total, 62 studies were selected and analyzed to address the three research questions. More specifically, the review examines how modern AI techniques are applied across SDLC activities, what impacts on software development are reported, and what challenges limit their effective adoption in practice. By bringing together evidence from different research streams, this study aims to provide a clear and structured cross-phase understanding of the evolving role of AI in software engineering.

The main contributions of this study are threefold. First, it provides a lifecycle-oriented synthesis of modern AI applications across software development activities. Second, it examines the stated impact of various technologies on productivity, artifact quality, and developer processes. Third, it outlines the primary limitations, threats, and adoption issues that affect the ethical integration of AI into software engineering practice.

The remainder of this study is organized as follows: Section II presents an overview of modern AI and the SDLC. Section III reviews related studies. Section IV discusses the research methodology. Section V provides results and addresses the research questions. Finally, Section VI concludes the study and outlines directions for future work.

## II. Background

In this review, *modern AI* primarily refers to generative AI systems applied to software development, with a particular focus on large language model (LLM)-based tools, AI agents, and agentic workflows. Generative Artificial Intelligence (GenAI) denotes AI systems capable of producing original artifacts, including natural language text, images, and source code, in response to prompts or other forms of input [1]. In software engineering, the most relevant form of GenAI is represented by LLMs, which are deep learning models based on the Transformer architecture [2] and pre-trained on large-scale corpora of natural language and source code using self-supervised learning objectives. This paradigm enables broad

*Corresponding author

capabilities such as code generation, explanation, summarization, and reasoning support without requiring task-specific supervision [3].

Three levels of modern AI are distinguished throughout this review, where each one is characterized by an increasing degree of autonomy and coordination complexity: LLMs as reactive generation systems, AI Agents as tool-augmented planning architectures, and Agentic AI as multi-agent collaborative systems operating with limited human oversight. Table I summarizes these distinctions.

TABLE I. OPERATIONAL CHARACTERISTICS OF MODERN AI LEVELS

| Level | Memory Type | Coordination | Role |
|---|---|---|---|
| LLM | Context-only | None | Text/code generation |
| AI Agent | Persistent | Single-agent | Tool-augmented task execution |
| Agentic AI | Shared | Multi-agent | Workflow orchestration |

While LLMs provide strong generative and reasoning capabilities, a standard model usually operates as a reactive system that maps a given input to an output. To support more structured and autonomous task execution, recent research increasingly embeds LLMs into AI agents [4]. These agents consist of a base model combined with persistent memory, access to external tools, and planning or control mechanisms that enable multi-step perception–planning–action loops [5]. In software development, an AI agent can perform functions like interacting with software artifacts, running code execution tools, obtaining documentation, and iteratively refining outputs. Most recently, this paradigm has expanded toward *agentic AI*, in which one or more AI agents collaborate to achieve complex goals with limited human supervision [6]. In this case we find an orchestrating layer which typically divides a main goal into subtasks, distributes them to specialized agents, and consolidates intermediate results into a final output [7]. Compared to a single-call LLM interaction, agentic systems introduce higher degrees of autonomy, coordination, and decision delegation, making them well-suited for complex software engineering workflows. Fig. 1 illustrates this conceptual progression.

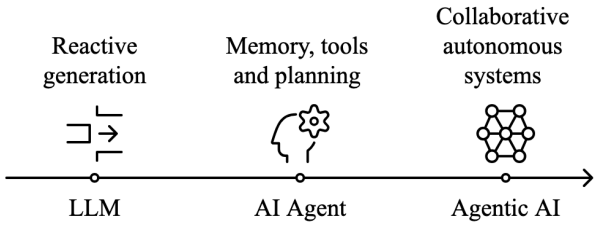


Fig. 1. Levels of modern AI in software development.

## III. RELATED WORK

Recent reviews have examined modern AI in software engineering from several perspectives. Broad syntheses have mapped LLM applications across development tasks and highlighted challenges related to evaluation reliability, hallucination, and human–AI workflows [8], [9], while more focused studies have investigated specific subdomains such as software testing, vulnerability detection, and requirements engineering [10], [11], [12]. In parallel, dedicated reviews have explored multi-agent and agentic approaches, emphasizing autonomy, task orchestration, and collaboration patterns [7], [13].

However, these studies tend to focus on particular technologies or development activities, offering valuable but partial perspectives. A synthesis connecting how modern AI is applied across the full software development lifecycle, what impacts are reported, and what limitations and adoption challenges emerge from current evidence is still needed.

Accordingly, this study addresses that gap through a systematic literature review organized around a four-phase lifecycle framework: Inception, Construction, Quality Assurance, and Operations. Different from previous research such as [8] and [9], which discuss the use of LLMs without a lifecycle structure, and [7] and [13], which focus on agentic architectures without spanning all SDLC phases, this review synthesises empirical evidence on modern AI applications, reported impacts, and adoption challenges through a unified four-phase lifecycle lens.

## IV. METHODOLOGY

The methodology section describes the systematic approach adopted in this study in order to ensure transparency, consistency, repeatability, and rigor in the review process. To achieve this, the review was conducted following the PRISMA guidelines [14] and aligns with the Kitchenham and Charters SLR guidelines for software engineering [15].

### A. Study Objective and Scope

The aim of this systematic literature review is to synthesize empirical studies that examine how modern AI is being used in software development. The collected evidence is organized around three main aspects: first how AI is used across the software development lifecycle, what impacts on development are reported, and what limitations and adoption challenges are identified. The review focuses only on primary research studies, while secondary sources such as surveys, literature reviews, and mapping studies are excluded.

This review covers papers published between 2023 and 2025. This period was mainly chosen in order to include research conducted after the public release of ChatGPT which was late 2022 [16], and which created great interest in generative AI and large language models and their application in software development. By restricting the review to this time period, the study only focuses on actual advances in modern AI rather than classical or prior AI approaches.

To structure the synthesis across software development activities, this review adopts a consolidated four-phase analytical framework comprising *Inception*, *Construction*, *Quality Assurance*, and *Operations*. This structure is partially inspired by emerging industry conceptualizations of AI-driven development lifecycles, such as the AI-Driven Development Life Cycle (AI-DLC) [17]. However, unlike such models, this review treats Quality Assurance as a distinct analytical phase in order to better capture validation-oriented activities such as testing, debugging, code review, and security verification. Fig. 2 illustrates this framework and the main categories of activities associated with each phase.

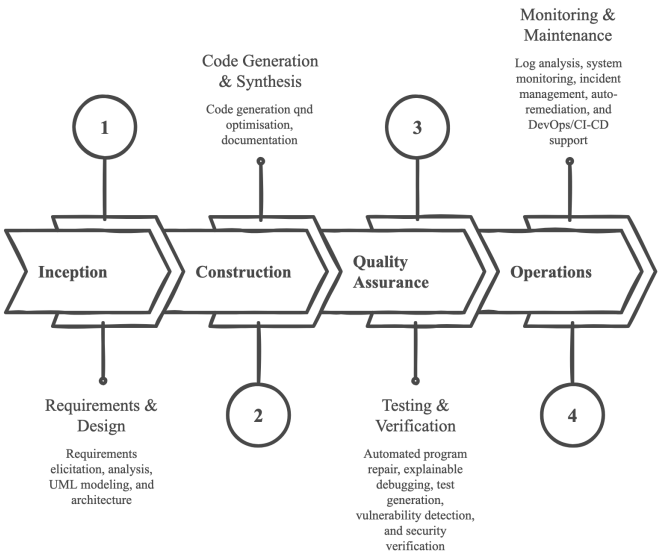


Fig. 2. Four-phase software development lifecycle framework adopted in the review.

TABLE II. PICOC FRAMEWORK DEFINING THE REVIEW SCOPE

| Dimension | Definition |
|---|---|
| Population | Software developers, software engineering teams, researchers, and software projects. |
| Intervention | Modern AI approaches including generative AI, LLMs, AI code assistants, AI agents, retrieval-augmented generation, and agentic workflows. |
| Comparison | Non-AI approaches or alternative AI-based techniques, where applicable. |
| Outcome | SDLC applications, reported impacts, limitations, and adoption challenges. |
| Context | Peer-reviewed English-language studies in software engineering, published between 2023 and 2025. |

To further define the scope and keywords used in our study we generated the PICOC framework presented in Table II.

### B. Research Questions

To structure the synthesis of recent empirical evidence, this review is guided by three research questions addressing the usage, impact, and constraints of modern AI in software development.

**RQ1: What are the current applications of modern AI in the SDLC?**

This question aims to identify the current state and trends in the application of modern AI techniques throughout the software development lifecycle. By analyzing reported applications in activities such as requirements engineering, implementation, testing, and operational support.

**RQ2: What are the reported impacts of modern AI on software development practices ?**

This focuses on reported changes related to productivity, development speed, artifact quality, collaboration dynamics, and decision support.

**RQ3: What limitations, risks and adoption challenges are perceived from the integration of modern AI in software development?**

This research question addresses the technical reliability issues, security and governance concerns, organizational barriers, and the new challenges introduced by LLMs, multi-agent and autonomous AI systems.

### C. Data Sources and Search Strategy

The literature search was mostly done on *Scopus*, which was used as the main data source due to its broad coverage of computer science topics. In order to achieve broader coverage of studies related to this research, *IEEE Xplore* and the *ACM Digital Library* were also used as they provide a vast amount of software engineering conference and journal papers. Preprint repositories such as *arXiv* were consulted to identify latest developments in generative AI, especially to inform the background and related work sections of this review. However, it should be noted that, only studies indexed in Scopus were retained in the final corpus, ensuring that all selected publications meet a peer-review quality standard [15].

The search string combined both modern AI and software development activity terms, as follows:

(*"generative AI" OR "large language model" OR "LLM" OR "AI agent" OR "agentic AI"*) AND (*"software development" OR "software engineering" OR "SDLC" OR "requirements engineering" OR "software architecture" OR "code generation" OR "software testing" OR "software maintenance" OR "DevOps"*)

The query was applied to the titles, abstracts, and keywords section. Additional filters were used to restrict results to the defined publication period (2023-2025), English language, document types corresponding to articles and conference papers, and the Computer Science subject area.

### D. Inclusion and Exclusion Criteria

To ensure consistency in study selection, predefined inclusion and exclusion criteria were applied throughout the screening process. The inclusion criteria are summarized in Table III, and the exclusion criteria are presented in Table IV.

### E. Study Selection Process

The study selection process followed a PRISMA-inspired workflow. The first phase of search involved using Scopus database and the predefined search string resulting in 5,336 records, which were then filtered to retain only English-language articles and conference papers published between 2023 and 2025 in the Computer Science subject area, reducing the corpus to 3,888 records.

These records were screened on the basis of titles and abstracts, retaining 399 studies for full-text assessment. Full-text screening was then conducted to verify compliance with the predefined inclusion and exclusion criteria and to confirm publication accessibility. After this final eligibility assessment, 62 studies were included in the review corpus, where 49 forming the main SDLC-oriented corpus used to address RQ1

TABLE III. INCLUSION CRITERIA BY SEARCH SUBJECT

| Search Subject | Inclusion Criteria |
|---|---|
| Modern AI Applications in Software Development | – Proposes or evaluates LLM-based, generative AI, or agentic approaches applied to software development activities.<br>– Addresses at least one SDLC phase: Inception, Construction, Quality Assurance, or Operations.<br>– Reports empirical evaluation, controlled experiment, or tool validation. |
| Impacts on Development Practices and Outcomes | – Measures or discusses productivity, output quality, or workflow changes resulting from AI adoption.<br>– Provides quantitative or qualitative evidence of AI impact on development speed, correctness, or developer support.<br>– Highlights strengths, limitations, and practical implications of the proposed approach. |
| Limitations, Risks, and Adoption Challenges | – Identifies technical, security, or organizational barriers to AI adoption in software engineering.<br>– Discusses reliability issues such as hallucination, non-determinism, or adversarial robustness.<br>– Focuses on governance, privacy, or socio-technical challenges in AI-assisted development.<br>– Includes specific findings on agentic AI risks such as cascading errors or coordination failures. |
| Publication Eligibility | – Published between 2023 and 2025.<br>– Peer-reviewed primary journal article or conference paper.<br>– Written in English and retrieved within the defined database search scope. |

TABLE IV. EXCLUSION CRITERIA BY SEARCH SUBJECT

| Search Subject | Exclusion Criteria |
|---|---|
| Scope and Domain | – Not focused on software development activities (e.g. healthcare, education, or finance).<br>– Modern AI is not the primary intervention or technique under study.<br>– Uses only traditional ML, rule-based systems, or classical NLP without LLMs or generative AI. |
| Empirical Evidence | – Lacks sufficient empirical evidence or without experimental validation.<br>– Secondary study such as a survey, systematic literature review, systematic mapping, or meta-analysis. |
| Publication Accessibility | – Full text not accessible for eligibility assessment or data extraction. |

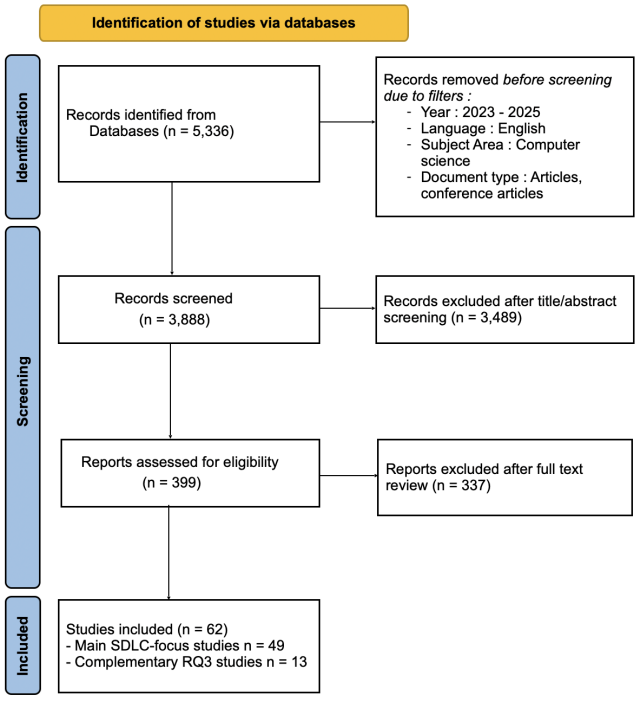


Fig. 3. PRISMA flow diagram for the review.

and RQ2, and 13 complementary studies included to strengthen the analysis of limitations, risks, and adoption challenges in RQ3. The overall selection process is summarized in Fig. 3.

## V. RESULTS AND DISCUSSION

### A. RQ1: Applications of Modern AI Across SDLC Phases

AI tools are currently employed throughout all stages of software development, from requirements gathering to system maintenance, as shown in Table V.

*1) Inception:* In Inception phase, AI is mostly used for requirements engineering, software modeling, and architectural assistance. LLMs help in generating requirements from conversations held with clients [21], refine backlog items and improve user story quality [19], [25], or detect ambiguities or inconsistencies in specifications [23], [29]. They can also help generate UML diagrams [22], [24], [26], and help novice analysts work with modeling tools [28]. Overall, AI in Inception mostly assists with drafting early outputs and formalizing requirements, whereas planning and estimation remain underexplored.

*2) Construction:* In Construction phase, AI is used for a wide range of tasks including writing code and generating documentation. Rather than asking a model to generate code from a simple bare prompt, studies show that methods like retrieval-augmented generation (RAG), guided prompting, or requirement-aware pipelines significantly improve the correctness of the generated code [34], [36], [38]. When developers need to handle more complex tasks, iterative refinement and multi-agent systems help improve results further [35], [45], [46]. Recent work also shows that AI can generate code directly from visual inputs such as flowcharts and UML diagrams [40], [41], and that LLMs can be embedded into CI/CD pipelines to automatically convert backlog items into functional code [39]. AI additionally supports documentation

TABLE V. SYNTHESIS OF REVIEWED STUDIES ACROSS SOFTWARE DEVELOPMENT PHASES.

| Ref. | Phase | Sub-theme | AI Approach |
|---|---|---|---|
| [18] | Inception | Requirements elicitation | GPT-4-Turbo agent simulation |
| [19] | Inception | Requirements elicitation | Multi-agent LLM system |
| [20] | Inception | Backlog classification | RoBERTa fine-tuning |
| [21] | Inception | Requirements from conversations | Llama-2 prompting |
| [22] | Inception | UML and code generation | LLM-driven MDA pipeline |
| [23] | Inception | SRS quality evaluation | GPT-4 zero-shot |
| [24] | Inception | UML sequence diagrams | GPT-3.5 prompting |
| [25] | Inception | User story quality | LLM agent prompting |
| [26] | Inception | UML class diagram modeling | GPT-4 evaluation |
| [27] | Inception | Domain model generation | ChatGPT-3.5 |
| [28] | Inception | UML modeling assistance | LLM support for novices |
| [29] | Inception | Requirements ambiguity detection | GPT-4 zero-shot |
| [30] | Inception | Requirements elicitation | ChatGPT prompting |
| [31] | Inception | Interview script generation | ChatGPT and Bard |
| [32] | Inception | Domain model extraction | GPT-3.5 vs CRF comparison |
| [33] | Inception | Architectural decision records | GPT-4 and Flan-T5 |
| [34] | Construction | RAG-based code generation | Retrieval-augmented generation |
| [35] | Construction | Iterative code refinement | Multi-role LLM pipeline |
| [36] | Construction | Guided code generation | BM25 retrieval prompting |
| [37] | Construction | Decoding configuration | Temperature tuning |
| [38] | Construction | Requirement-to-code generation | Requirement-aware prompting |
| [39] | Construction | Backlog-to-code pipeline | GPT-4 CI/CD integration |
| [40] | Construction | Flowchart-to-code generation | Multimodal LLMs |
| [41] | Construction | UML-to-code generation | GPT-4 Vision |
| [42] | Construction | REST API endpoint extraction | LLM agents |
| [43] | Construction | Use case documentation | Structured prompting |
| [44] | Construction | Code summarization | LLM-based summarization |
| [45] | Construction | End-to-end agentic development | Multi-agent framework (MetaGPT) |
| [46] | Construction | End-to-end agentic development | Chat chain protocol (ChatDev) |
| [47] | Quality Assurance | Automated program repair | LLM & static analysis |
| [48] | Quality Assurance | Automated program repair | Retrieval-augmented repair |
| [49] | Quality Assurance | Multi-method bug repair | Divide-and-conquer agents |
| [50] | Quality Assurance | Explainable debugging | Scientific debugging prompting |
| [51] | Quality Assurance | Vulnerability-witnessing tests | GPT-4 Turbo |
| [52] | Quality Assurance | Test quality assessment | GPT-4o and Amazon Q |
| [53] | Quality Assurance | Automated test generation | State space models |
| [54] | Quality Assurance | Smart contract vulnerability | GPT-3.5 fine-tuned |
| [55] | Quality Assurance | Smart contract vulnerability | Multi-agent audit system |
| [56] | Quality Assurance | Industrial code review | Fine-tuned LLM (BitsAI-CR) |
| [57] | Quality Assurance | Code review automation | Prompt engineering vs fine-tuning |
| [58] | Operations | Cloud observability | Knowledge graph & LLM |
| [59] | Operations | Network monitoring | Llama-3.2 multi-agent |
| [60] | Operations | Incident root cause analysis | Hierarchical multi-agent (SRE) |
| [61] | Operations | Maintenance assistance | Ontology-based RAG |
| [62] | Operations | Incident mitigation | Fine-tuned Code-davinci |
| [63] | Operations | Automated root cause analysis | GPT-4 workflow-driven (RCACopilot) |
| [64] | Operations | Log parsing | Flan-T5 and LLaMA few-shot |
| [65] | Operations | Auto-remediation | GPT-4 Ansible playbook generation |
| [66] | Operations | CI/CD pipeline support | Workflow-integrated LLM |

tasks such as generating use case descriptions and summarizing source code [43], [44], while LLM-based agents have been used to extract REST API endpoints automatically from microservice architectures [42].

*3) Quality assurance:* In Quality Assurance phase, AI is mainly used for code review, automated testing, vulnerability recognition, debugging, and software repair. Pairing LLMs with static analysis [47], retrieval [48], or multi-agent decomposition [49] results in more reliable bug fixes compared to direct generation. AI debugging tools assist developers in determining whether a fix is correct [50]. AI may generate syntactically valid test cases [51], [52], but without class-level dependencies or runtime data, the tests frequently lack semantic correctness [53]. In order to improve vulnerability detection fine-tuned [54] and multi-agent [55] approaches can be used in domains such as smart contracts, but generalization to larger codebases remains limited. In code review, the use of prompt engineering techniques [56] and fine-tuning [57] provide more meaningful and reliable feedback than basic prompting.

*4) Operations:* In Operations phase, AI can assist in doing log analysis, giving incident diagnosis and maintenance guidance. Systems based on LLM can semantically interpret logs [64]. Now, incident management has progressed from single-model diagnostic support [62] to multi-agent [59], [60] and workflow-driven systems [63] that integrate logs, metrics, and traces to identify root causes and suggest solutions. In addition, AI can also generate remediation scripts and playbooks through maintenance assistants [61] and auto-remediation frameworks [65]. More recently, LLMs can also help developers work with GitHub repositories by reading configuration files and supporting CI/CD decisions [66]. Overall, AI in Operations is acting as an intelligent interpretation and action layer, though scalability, safety, and performance in complex production environments remain open challenges.

Across phases, the role of AI in software engineering has gradually increased, moving from helping structure early outputs in Inception, to generating and coordinating code in the Construction stage, supporting verification in Quality Assurance, and taking action in Operations. However, fully autonomous AI remains currently out of reach due to limitations in semantic reasoning, domain knowledge, and integration complexity.

Looking across the reviewed studies, several patterns emerge beyond what Table V individually captures. While contributions are present across all lifecycle phases, their distribution within each phase is uneven. In Inception, studies are heavily concentrated on requirements elicitation and UML modeling, leaving activities such as project planning and effort estimation largely unaddressed. Construction and Quality Assurance attract the broadest range of contributions, as tasks like code generation, program repair, and testing are naturally suited to automation. Operations, despite growing interest, remains the least explored phase, particularly regarding runtime safety and long-term system maintenance.

In addition to these distributions, certain patterns in the effectiveness of modern AI methodologies have been observed. For example, the use of retrieval-augmented generation (RAG) generally increases the accuracy of output compared to zero-shot or basic prompting methods, especially in tasks related to code generation and repair due to the fact that it grounds model outputs in project-specific context and reduces the chances of hallucination. Furthermore, the use of multi-agent and agentic systems demonstrate better performance in complex, multi-step workflows. However, such systems need more coordination and suffer from a higher risk of cascading mistakes. Finally, fine-tuning provides smoother and specialized performance especially when it comes to code reviewing and vulnerability detection but involves high resource expenses. Overall, none of the discussed methods outperforms others in all cases, and the efficiency of techniques depends greatly on the complexity and nature of the task.

### B. RQ2: Reported Impacts on Software Development Practices and Outcomes

The reviewed studies report three main categories of impact associated with the adoption of modern AI: faster development, more consistent outputs, and changes in how development teams work.

The most commonly reported benefit of modern AI adoption in software development is increasing productivity, as it helps reduce manual work in requirements elicitation [18], [19] and backlog management [25]. AI also produces more correct code especially when used with retrieval [34], guided prompting [36], or generation guided by project requirements [38]. In Quality Assurance and Operations, AI speeds up program repair [47], code review [56], incident diagnosis [60], log parsing [64], and root cause recommendations [62]. That said, these gains are task-dependent and tend to drop in complex or open-ended activities [23], [40], [53].

Multi-agent systems also changed how development teams work, as they support task decomposition [45], [46] and connect AI to CI/CD and operational workflows [65], [66]. LLM tools can also help junior developers with modeling [28], debugging [50], and incident response [61].

As for output quality, AI tools provide better results particularly in requirements [24], documentation [43], [44], and code review [56]. However, a recurring issue that has been found in the literature is that these outputs are often syntactically correct but semantically wrong, meaning that the code compiles and follows proper syntax rules but does not actually do what was intended or fails to capture the required behavior, especially in testing [51] and vulnerability detection [54], [55]. This is an issue that persists across all lifecycle phases and is addressed by introducing several techniques. Fig. 4 summarizes the three main approaches repeatedly reported in the reviewed studies: prompt engineering, retrieval-augmented generation, and fine-tuning. Prompt engineering structures the model's reasoning before it generates, leading to more accurate code [36], better debugging support [50], and more useful code review feedback [56], then we have Retrieval-augmented generation that grounds the model in relevant existing code or documentation, with improvements reported in code generation [34], program repair [48], and maintenance assistance [61]. Finally, fine-tuning, which is a technique that adjusts the model according to the task or domain, enhancing its effectiveness for software repair [49], automated code review [57], and vulnerability detection [54]. Together, these three techniques consistently outperform basic zero-shot generation by highlighting the fact that raw model capability alone is insufficient for AI-assisted development.

Taken together, the literature suggests that modern AI contributes to incremental yet systemic shifts in software development practices. While productivity acceleration and artifact standardization are increasingly observable, fully autonomous development remains constrained by semantic reasoning limitations, domain specificity, and integration challenges.

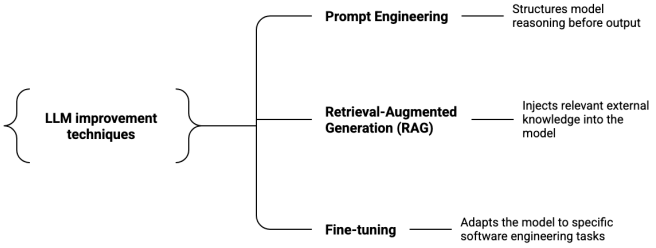


Fig. 4. LLM-based techniques for improving output quality in AI-assisted software development.

### C. RQ3: Limitations, Risks, and Adoption Challenges

A broad set of limitations and risks has been encountered in the literature, as summarized in Table VI. These challenges span technical reliability issues, architectural constraints, and socio-technical adoption barriers, and are further amplified in emerging agentic development paradigms.

TABLE VI. COMPLEMENTARY RQ3 STUDIES

| Ref. | Focus | Risk Category |
|---|---|---|
| [67] | GenAI adoption in organizations | Governance |
| [68] | Uncertainty in code completions | Automation bias |
| [69] | LLM self-verification | Self-contradiction |
| [70] | Correctness via internal states | Reasoning limits |
| [71] | Data race detection | Runtime visibility |
| [72] | Security of AI-generated code | Vulnerabilities |
| [73] | LLM security awareness | Passive security risk |
| [74] | Membership inference attacks | Privacy leakage |
| [75] | Adversarial code completion | Robustness |
| [76] | Multi-agent process modeling | Coordination errors |
| [77] | Debate-verify-debug pipelines | Cascading errors |
| [78] | LLM agent security paradigms | Chained attacks |
| [79] | End-to-end multi-agent generation | Autonomy risks |

From a technical perspective, hallucination and non-determinism are the most consistently reported limitations across SDLC phases. As discussed in RQ2, AI-generated outputs are frequently syntactically plausible yet semantically incorrect, affecting requirements modeling [23], code generation [35], program repair [49], and operational diagnosis [58]. Variability in outputs for identical inputs complicates reproducibility and quality assurance, while decoding configuration [37] and prompt sensitivity [44] further increase deployment complexity. Adversarial robustness also remains limited: recent studies demonstrate that malicious natural-language instructions can significantly increase the likelihood of insecure code generation while preserving functional correctness [75]. In agentic settings, cascading errors across multi-step pipelines introduce additional instability, as early planning inaccuracies propagate through downstream development stages [38], [77], [79].

Architectural and resource constraints represent a second category of limitations. Context window restrictions hinder cross-file reasoning in requirements analysis [29], program repair [47], and vulnerability detection [54], leading most studies to evaluate AI performance on simplified task units. Multi-agent systems also come with a lot of costs: they consume more tokens, introduce coordination delays, and require careful

orchestration, both in agentic development frameworks [45] and operational environments [60]. Approaches such as fine-tuning [57], retrieval integration [34], and hybrid verification pipelines [53] further demand significant computational resources and labelled data, which limits how widely they can be adopted. Another gap is that most current AI tools work on static code or logs and cannot observe system behavior at runtime, leaving dynamic issues [71] and vulnerabilities that only appear during execution [55] largely undetected.

Security and organizational challenges add another layer of concern. AI-generated code frequently contains vulnerabilities [72], and models rarely flag security issues on their own unless the user explicitly asks [73]. There is also a growing risk around privacy where attackers can use membership inference techniques to extract information about the data used to train a model, potentially exposing proprietary code or sensitive information [74]. In agentic systems, the risks go further — chained attacks that combine prompt injection with other exploitation techniques can manipulate how agents use tools or communicate with each other, creating new threats that traditional security models were not designed to handle [78].

Aside from these technological issues, organizational challenges also arise. Accepting AI suggestions without verification can become a harmful habit for developers [68]. Also, there is no clear legal framework governing how AI tools handle sensitive code or data [67]. While AI can assist junior engineers learn and speed up repetitive processes, over-reliance on low-quality or imprecise outputs [56], [61] might silently decrease the standard of the final result.

## VI. Conclusion and Future Work

In conclusion, this review shows that current software development tasks tend to incorporate modern artificial intelligence. In accordance with the reviewed literature, not only does AI play the role of providing individual help, but it tends to assist in development processes overall. It is considered as an infrastructure layer that facilitates developers' work on various software development stages. Its advantages are primarily perceived in automated, repetitive and structured actions. However, there are still considerable challenges to AI integration, such as security problems, privacy protection, context management limitations, and other factors.

For practitioners, this review highlights various insights that can actually be taken into consideration. In this regard, prompt engineering, RAG, and fine-tuning consistently beat zero-shot approaches in terms of performance and need to be considered an integral part of the development process and not just additional features. As for multi-agent systems, they provide performance boosts for more intricate activities, but come with coordination costs and error propagation, which need proper management.

Future studies should focus on various gaps found in the current literature, most specifically in project planning, CI/CD pipeline automation, repository-level development, and vulnerability detection. The effects of AI adoption on developer skills, team structures, and engineering culture also presents a relevant investigation. As multi-AI agents and agentic AI systems become more and more common, we should also consider how agent teams should be organized and scaled as task complexity increases, on whether AI can support genuine creativity and innovation beyond pattern reproduction, and how autonomous agents can make reliable decisions when outputs conflict or ambiguities arise. Finally, as agentic AI becomes more capable, advancements in abilities must be accompanied by equally significant improvements in safety, accountability, and governance.

## References

[1] S. Feuerriegel, J. Hartmann, C. Janiesch, and P. Zschech, "Generative AI," *Business & Information Systems Engineering*, vol. 66, pp. 111–126, 2024.

[2] A. Vaswani et al., "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[3] C. Ebert and P. Louridas, "Generative AI for Software Practitioners," *IEEE Software*, vol. 40, no. 4, pp. 30–38, 2023.

[4] L. Wang et al., "A Survey on Large Language Model Based Autonomous Agents," *Frontiers of Computer Science*, vol. 18, no. 6, 2024.

[5] Y. Liu et al., "Agent Design Pattern Catalogue," *Journal of Systems and Software*, 2025.

[6] IBM, "What is Agentic AI?," 2024.

[7] J. He, C. Treude, and D. Lo, "LLM-Based Multi-Agent Systems for Software Engineering," *ACM Transactions on Software Engineering and Methodology*, 2025.

[8] X. Hou et al., "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Transactions on Software Engineering and Methodology*, 2024.

[9] A. Fan et al., "Large Language Models for Software Engineering: Survey and Open Problems," in *Proc. ICSE-FoSE*, 2023.

[10] J. Wang et al., "Software Testing with Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, 2024.

[11] X. Zhou et al., "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," *ACM Transactions on Software Engineering and Methodology*, 2025.

[12] A. Hemmat et al., "Research Directions for Using LLM in Software Requirement Engineering: A Systematic Review," *Frontiers in Computer Science*, 2025.

[13] N. Akbar et al., "Methods and Techniques of Agentic Software Engineering: A Systematic Literature Review," *IEEE Access*, 2026.

[14] M. J. Page et al., "The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews," *BMJ*, vol. 372, 2021.

[15] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," EBSE Technical Report EBSE-2007-01, 2007.

[16] OpenAI, "ChatGPT: Optimizing Language Models for Dialogue," OpenAI Blog, Nov. 2022.

[17] S. P. Raja and A. Mishra, "AI-Driven Development Life Cycle (AI-DLC)," AWS DevOps Blog, 2025.

[18] Ataei et al., "Elicitron: An LLM Agent-Based Simulation Framework for Design Requirements Elicitation," 2024.

[19] Sami et al., "Early Results of an AI Multiagent System for Requirements Elicitation and Analysis," 2024.

[20] Van Can and Dalpiaz, "Locating Requirements in Backlog Items: Content Analysis and Experiments with Large Language Models," 2025.

[21] Voria et al., "RECOVER: Toward Requirements Generation From Stakeholders' Conversations," 2025.

[22] Babaalla et al., "LLM-Driven MDA Pipeline for Generating UML Class Diagrams and Code," 2025.

[23] Krishna et al., "Using LLMs in Software Requirements Specifications: An Empirical Evaluation," 2024.

[24] Ferrari et al., "Model Generation with LLMs: From Requirements to UML Sequence Diagrams," 2024.

[25] Zhang et al., "LLM-based Agents for Automating the Enhancement of User Story Quality: An Early Report," 2024.

[26] De Bari et al., "Evaluating Large Language Models in Exercises of UML Class Diagram Modeling," 2024.

[27] Bozyigit et al., "Generating Domain Models from Natural Language Text Using NLP: A Benchmark Dataset and Experimental Comparison of Tools," 2024.

[28] Wang et al., "How LLMs Aid in UML Modeling: An Exploratory Study with Novice Analysts," 2024.

[29] Mahbub et al., "Can GPT-4 Aid in Detecting Ambiguities, Inconsistencies, and Incompleteness in Requirements Analysis? A Comprehensive Case Study," 2024.

[30] Ronanki et al., "Investigating ChatGPT's Potential to Assist in Requirements Elicitation Processes," 2023.

[31] Görer and Aydemir, "Generating Requirements Elicitation Interview Scripts with Large Language Models," 2023.

[32] Arulmohan et al., "Extracting Domain Models from Textual Requirements in the Era of Large Language Models," 2023.

[33] Dhar et al., "Can LLMs Generate Architectural Design Decisions? An Exploratory Empirical Study," in *IEEE International Conference on Software Architecture (ICSA)*, 2024.

[34] Yang et al., "An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities," *ACM Transactions on Software Engineering and Methodology*, 2025.

[35] Chang et al., "A Self-Iteration Code Generation Method Based on Large Language Models," 2024.

[36] Li et al., "AceCoder: An Effective Prompting Technique Specialized in Code Generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[37] Arora et al., "Optimizing LLMs for Code Generation: Which Hyperparameter Settings Yield the Best Results?," 2024.

[38] Han et al., "ARCHCODE: Incorporating Software Requirements in Code Generation with Large Language Models," 2024.

[39] Sarschar et al., "PACGBI: A Pipeline for Automated Code Generation from Backlog Items," in *ASE*, 2024.

[40] He et al., "Flow2Code: Evaluating Large Language Models for Flowchart-based Code Generation Capability," 2024.

[41] Antal et al., "Toward a New Era of Rapid Development: Assessing GPT-4-Vision's Capabilities in UML-Based Code Generation," in *LLM4Code*, 2024.

[42] Chaplia and Klym, "Extracting REST API Endpoints from Microservices Using LLM Agents," in *DESSERT*, 2024.

[43] Naimi et al., "Automating Software Documentation: Employing LLMs for Precise Use Case Description," in *KES*, 2024.

[44] Sun et al., "Source Code Summarization in the Era of Large Language Models," 2024.

[45] Hong et al., "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," in *ICLR*, 2024.

[46] Qian et al., "ChatDev: Communicative Agents for Software Development," in *ACL*, 2024.

[47] Li et al., "Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis," 2024.

[48] Liu et al., "ReAPR: Automatic Program Repair via Retrieval-Augmented Large Language Models," *Software Quality Journal*, 2025.

[49] Xie et al., "PReMM: LLM-Based Program Repair for Multi-method Bugs via Divide and Conquer," in *OOPSLA*, 2025.

[50] Kang et al., "Explainable Automated Debugging via Large Language Model-driven Scientific Debugging," 2023.

[51] Antal et al., "Leveraging GPT-4 for Vulnerability-Witnessing Unit Test Generation," in *EASE*, 2025.

[52] Alves et al., "Quality Assessment of Python Tests Generated by Large Language Models," 2024.

[53] Iznaga et al., "Integrating Large Language Models into Automated Software Testing," *Future Internet*, 2025.

[54] Liu et al., "Exploring the Potential of ChatGPT in Detecting Logical Vulnerabilities in Smart Contracts," *Blockchain: Research and Applications*, 2025.

[55] Wei et al., "Advanced Smart Contract Vulnerability Detection via LLM-Powered Multi-Agent Systems," *IEEE Transactions on Software Engineering*, 2025.

[56] Sun et al., "BitsAI-CR: Automated Code Review via LLM in Practice," 2024.

[57] Pornprasit and Tantithamthavorn, "Fine-tuning and Prompt Engineering for LLMs-based Code Review Automation," *Information and Software Technology*, 2024.

[58] Amanmadov and Abdullayev, "From Logs to Knowledge: LLM-Powered Dynamic Knowledge Graphs for Real-Time Cloud Observability," 2025.

[59] Yoon et al., "OFCnetLLM: Large Language Model for Network Monitoring and Alertness," 2025.

[60] Kataria, "Intelligent Site Reliability Engineering: A Multi-agent LLM Framework for Automated Incident Analysis and Root Cause Determination," *IJES*, 2025.

[61] Ludwig et al., "An Ontology-based Retrieval Augmented Generation Procedure for a Voice-Controlled Maintenance Assistant," *Computers in Industry*, 2025.

[62] Ahmed et al., "Recommending Root-Cause and Mitigation Steps for Cloud Incidents using Large Language Models," in *ICSE*, 2023.

[63] Chen et al., "Automatic Root Cause Analysis via Large Language Models for Cloud Incidents (RCACopilot)," in *EuroSys*, 2024.

[64] Ma et al., "LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing," in *ICSE*, 2024.

[65] Sarda et al., "Leveraging Large Language Models for the Auto-remediation of Microservice Applications," in *FSE Industry Track*, 2024.

[66] Zhang et al., "On the Effectiveness of LLMs for GitHub Workflows," 2024.

[67] Kemell et al., "Adoption of Generative AI in Software Organizations: A Multiple Case Study," 2025.

[68] Vasconcelos et al., "Uncertainty Highlighting for AI Code Completions," 2024.

[69] Yu et al., "Fight Fire With Fire: Self-Verification Mechanisms in LLM-Based Code Generation," 2024.

[70] Bui et al., "Correctness Assessment of AI-Generated Code via Internal Representations," 2024.

[71] Alsofyani and Wang, "Evaluating ChatGPT for Data Race Detection," 2024.

[72] Tihanyi et al., "How Secure is AI-Generated Code?," 2024.

[73] Sajadi et al., "Do Large Language Models Consider Security? An Empirical Study," 2024.

[74] Yang et al., "GOTCHA: Membership Leakage Attacks Against Code Models," 2023.

[75] Wan et al., "Adversarial Attacks on Code Completion Systems via Natural Language Injection," 2024.

[76] Lin et al., "MAO: Multi-Agent Orchestration for Process Modeling," 2025.

[77] Islam et al., "DVDCoder: Debate-Verify-Debug Multi-Agent Framework for Code Generation," 2024.

[78] Gasmi et al., "Security Analysis of LLM Agent Systems: MCP vs Function Calling," 2025.

[79] Mao et al., "Blueprint2Code: A Multi-Agent Framework for End-to-End Software Development," 2025.