Journal of Emerging Technologies with Industrial Applications
https://www.sciltp.com/journals/jetia

Review

Generative AI for Software Engineering: A Review of Code Generation, Testing, Human–AI Collaboration, and Quality Assurance

Yang Lu

Faculty of Business, Information & Human Sciences (FBIHS), Kuala Lumpur University of Science & Technology (KLUST), Kajang 43000, Malaysia; ziiyuu@gmail.com

How To Cite: Lu, Y. Generative AI for Software Engineering: A Review of Code Generation, Testing, Human–AI Collaboration, and Quality Assurance. *Journal of Emerging Technologies with Industrial Applications* **2026**, *1*(2), 10. https://doi.org/10.53941/jetia.2026.100010

Received: 22 July 2026
Revised: 21 August 2026
Accepted: 30 October 2026
Published: 5 November 2026

Abstract: Generative AI is increasingly reshaping software engineering (SE) across the development lifecycle. This review synthesises 113 peer-reviewed journal studies and organises the literature around five research questions and four application areas: code generation, testing and debugging, human–AI collaboration, and quality assurance. The reviewed studies show substantial progress in natural-language-to-code generation, context-aware programming, automated testing, fault localisation, program repair, and AI-assisted development. At the same time, persistent challenges remain in semantic correctness, non-deterministic outputs, software security, maintainability, verification, and evaluation consistency. Drawing these findings together, the paper presents a lifecycle mapping, a closed-loop AI-driven software engineering framework, and a four-dimension taxonomy to organise existing research and clarify connections across previously fragmented areas. Overall, generative AI is moving software engineering from a largely deterministic and human-centred workflow toward a more probabilistic and AI-enhanced engineering process. Further progress will depend on stronger verification, security, reliability, maintainability, and appropriately calibrated human oversight.

Keywords: generative AI; software engineering; code generation; software testing; human–AI collaboration; quality assurance

1. Introduction

Modern digital infrastructure rests, to a considerable degree, on software engineering, the discipline that underpins the construction of large and increasingly intricate applications in nearly every sector. As systems grow, so do the cost of building them and the burden of keeping them running [1]. Much of this work remains manual: design decisions, implementation, testing and the ongoing care of system behaviour all fall to developers. Methodologies such as Agile and DevOps have matured and do raise productivity, yet neither removes the dependence on scarce human expertise [2]. Requirements, moreover, rarely stay fixed—shifting demands inject uncertainty into a process that is already difficult to plan [3]. Therefore, software engineering is still a labour-intensive and error-prone discipline [4].

As a transformative technology, Generative AI has recently emerged in the field of software engineering. Large language models such as GPT, Codex and CodeBERT have shown strong capabilities in code generation and understanding. These models are built based on the Transformer architecture and trained using large-scale data sets [5]. The latest research published in journals such as IEEE Access, FGCS and Neurocomputing shows that generative artificial intelligence can support a variety of software engineering tasks, including code synthesis, test generation, defect detection and requirements analysis [6,7]. This marks the shift of software engineering from

Copyright: © 2026 by the authors. This is an open access article under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).
Publisher's Note: Scilight stays neutral with regard to jurisdictional claims in published maps and institutional affiliations

a rule-based system to a data-driven intelligent model [4]. Unlike traditional programming tools, generative models do not rely on predefined rules, but learn statistical patterns from large-scale data sets. This enables it to be generalised across multiple programming languages and software fields [8].

The rapid development of generative artificial intelligence has brought new opportunities to software engineering. One of the main driving forces is the automation of programming tasks: generative models can generate source code based on natural language descriptions, thus reducing manual workload [5]. Another driving force lies in the improvement of software testing and debugging: existing research shows that artificial intelligence systems can generate test cases and identify software defects [9,10], thus improving system reliability and shortening the development cycle [7]. In addition, human-computer collaboration is becoming an important research direction; developers interact with artificial intelligence systems more and more frequently in the process of software development [2], which changes the traditional division of roles and introduces a new development paradigm [11]. Finally, for systems generated by artificial intelligence, software quality assurance becomes more complex; how to ensure the correctness, security and reliability of the generated code is still a serious challenge [12].

Although remarkable progress has been made, there are still some research gaps in the existing literature. First of all, there is a lack of a unified framework to integrate Generative AI into the whole life cycle of software engineering; existing research often focusses on isolated tasks such as coding or testing [1]. Secondly, the current research lacks a comprehensive classification system that can relate to different application fields; code generation, testing, collaboration and quality assurance are usually studied separately, and there is a lack of unified analysis [13]. Third, the evaluation method of generative artificial intelligence in the field of software engineering is not yet unified; existing indicators (such as pass@k—the probability that at least one of k generated candidates passes the specified tests—or code coverage) cannot comprehensively measure the correctness and reliability of software [14]. Finally, people have limited understanding of the long-term impact of generative artificial intelligence on software engineering practice, which covers changes in developer behaviour, system design and maintenance process [15].

This research focusses on five research questions, aiming to build a systematic understanding of generative artificial intelligence in software engineering.

RQ1. How can generative AI be systematically integrated across the software engineering lifecycle within a unified framework?

RQ2. What are the primary approaches and unresolved challenges in AI driven code generation?

RQ3. How does generative AI contribute to automated software testing and debugging?

RQ4. In what ways does generative AI reshape human AI collaboration and developer roles?

RQ5. What are the key quality assurance concerns for AI generated software systems?

This study conducts a structured and comprehensive analysis of generative artificial intelligence in the field of software engineering [13] (Figure 1). The research contribution covers three levels: concept, method and analysis. At the conceptual level, this study introduces the unified life cycle perspective of software engineering under the background of generative intelligence [4], and integrates fragmented research into a coherent framework consistent with TSE and JSS literature [16]. At the methodological level, this study has built a classification system covering code generation, testing, collaboration and quality assurance [5], so that it can make a structured comparison of different research schools and be consistent with the classification method of EMSE [13]. At the analytical level, this study synthesises evidence from major peer-reviewed journals such as TSE, JSS, IST, EMSE, IEEE Access, FGCS and Neurocomputing [5,17]. This comprehensive analysis focusses on mechanism rather than isolated findings, which is consistent with the practice of empirical software engineering research [18]. In addition, this study also points out key research gaps, including evaluation inconsistency, reliability issues and limitations in human-computer collaboration understanding [14,19]; these gaps have been widely recognised in EMSE and IST literature [15]. Finally, this study outlines the future research direction, including autonomous software engineering, verifiable code generation and people-oriented AI systems [4,20], which are in line with the emerging trends in TSE and FGCS research [21].

Recent surveys have made important contributions, but most adopt a narrower task- or technology-centred focus. Jiang et al. [5] concentrate on LLM-based code generation, Faraji and Pombo [7] on AI-driven test automation, Zheng et al. [13] on LLM performance across software engineering tasks, and Otoum and Elkhalili [20] on agentic software engineering. The present review does not seek to replace these specialised surveys. Its distinctive role is to connect four application areas—code generation, testing and debugging, human–AI collaboration, and quality assurance—within a common lifecycle view, while also relating technical progress to developer roles, security and governance.

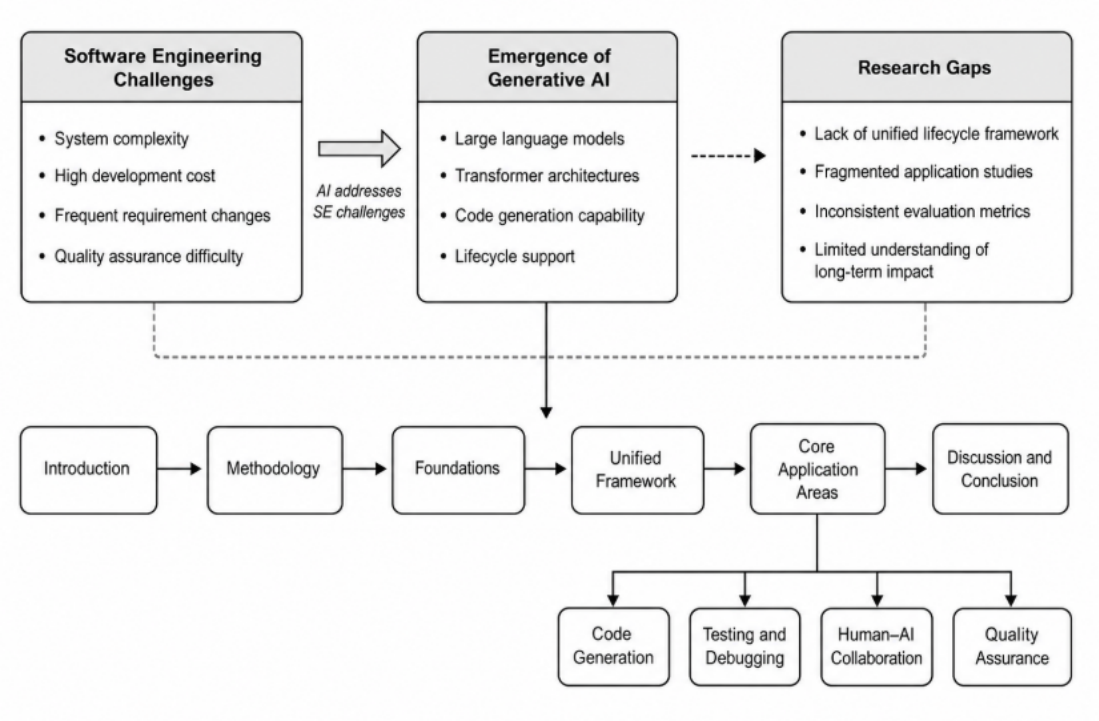


Figure 1. Research Motivation and Paper Structure.

2. Research Methodology

2.1. Data Sources and Search Strategy

The literature examined here was gathered through a systematic search of four mainstream academic databases—Web of Science, IEEE Xplore, the ACM Digital Library and Scopus. The choice was a practical one: between them, these services index the bulk of peer-reviewed work at the intersection of software engineering and artificial intelligence, and both communities rely on them routinely.

The search keywords are based on the cross-field combination of generative artificial intelligence and software engineering, covering terms such as generative artificial intelligence, large language model, code generation, software testing, debugging and software quality assurance. Boolean operators were then used to expand and combine these keywords, so that the search would capture relevant work as completely and as precisely as the databases allow.

Publication dates were restricted to the window from 2018 to 2026—a span that brackets the arrival of Transformer-based architectures and the subsequent surge of large language model research within software engineering [5,13].

The search string, in its final form, read as follows.

TS=((“Generative AI” OR “Large Language Models” OR “LLMs” OR “Foundation Models”)AND(“Software Engineering” OR “Software Development” OR “Software Lifecycle”)AND(“Code Generation” OR “Program Synthesis” OR “Software Testing” OR “Debugging”OR “Human-AI Collaboration” OR “AI-assisted Programming” OR “Software Quality Assurance”))

2.2. Inclusion and Exclusion Criteria

Strict inclusion criteria were applied to protect the quality of the corpus. The selected research must be a peer-reviewed journal article focused on the application of generative artificial intelligence in the field of software engineering. In addition, each study needs to cover at least one stage of the software engineering life cycle, such as code generation, testing, debugging, collaboration or quality assurance. Exclusion rules were defined alongside them to filter out work that was unrelated or of doubtful rigour: preprints, technical reports and other material that had not passed peer review were removed, as were studies with no genuine software engineering application. What survived this screening is a comparatively clean body of literature, and it is on this foundation that the later analysis and the classification scheme were built.

Most of the retained studies appeared in high-impact venues in software engineering and artificial intelligence (see Table 1), among them IEEE Transactions on Software Engineering, the Journal of Systems and Software,

Information and Software Technology, Empirical Software Engineering, IEEE Access, Neurocomputing and Future Generation Computer Systems [1,6]. Their selection reflects both their standing in the field and the breadth of theoretical and empirical work on generative artificial intelligence that they have carried [20,22].

The journal-focused scope is deliberate and should not be read as a judgement that peer-reviewed conference research is less rigorous. In software engineering, major conferences such as ICSE, ESEC/FSE, ASE and ISSTA are primary outlets for fast-moving work. Expanding the corpus to conference proceedings would require a new search and screening cycle and would materially change the evidence base reported here. The present review therefore retains the 113-study journal corpus, while treating the omission of conference literature as an explicit limitation (Section 9.8).

The final corpus comprises 113 journal studies. Although the search window began in 2018, the studies that satisfied all inclusion criteria and entered the final corpus were published between 2024 and 2026. Specifically, the final corpus comprises 16 studies from 2024, 52 from 2025, and 45 from 2026. The retained review record documents the databases, publication window, search string and eligibility rules, but it does not contain a PRISMA-style stage-by-stage count, an exact search execution date, or an inter-rater agreement statistic. Rather than reconstructing such values retrospectively, this revision makes that reporting boundary explicit and treats the review as a structured literature synthesis with transparent search parameters, while acknowledging the resulting reproducibility limitation in Section 9.8.

Table 1. Distribution of Selected Studies Across Journals.

Journal	Abbreviation	Count
IEEE Access	IEEE Access	33
IEEE Transactions on Software Engineering	TSE	23
ACM Transactions on Software Engineering and Methodology	TOSEM	17
Journal of Systems and Software	JSS	15
Empirical Software Engineering	EMSE	11
Information and Software Technology	IST	11
Neurocomputing	Neurocomputing	2
Future Generation Computer Systems	FGCS	1
Total		113

2.3. Classification Framework Design

This study adopts a multi-layer classification framework to systematically analyse the selected literature. The framework is based on the life cycle stage of software engineering and the construction of generative artificial intelligence capabilities.

The first layer represents the life cycle perspective of software engineering, covering requirements analysis, system design, code implementation, testing, deployment and maintenance [1]. This structure is widely used in software engineering literature.

The second layer represents the perspective of artificial intelligence capabilities, including natural language understanding, code generation, reasoning and verification support [13]. This layer is used to portray the functional capabilities of generative models in software engineering tasks.

The third layer represents the application scenario perspective, which includes four categories: code generation, testing and debugging, human-computer collaboration and software quality assurance [6]. This structure is consistent with the empirical research framework adopted in the recent systematic review [7].

The framework is deductive and synthesis-oriented. The lifecycle layer follows established software engineering stages, while the application layer reflects recurring clusters used across the reviewed literature. These categories are used to organise and compare the 113 studies; they were not generated through grounded-theory coding and are not claimed as an externally validated universal taxonomy. Accordingly, the framework should be interpreted as an analytical lens for this review rather than as a validated classification standard.

3. Foundations of Generative AI for Software Engineering

Generative artificial intelligence has evolved from an early statistical model to a Transformer-based architecture, which can capture the long-distance dependence between code and natural language [5], and thus give rise to universal and code-specific basic models [23]. However, systematic evaluations show that these models still face fundamental limitations in software engineering tasks [13]. This chapter will sort out this evolution process, outline its core competencies, and introduce the representative models that support current research.

3.1. Evolution of Generative AI

Generative artificial intelligence has gone through many different technical stages, and its ability to model natural language and programming structure has also gradually improved [5]. The earliest statistical approaches did little more than predict the probability of token sequences—language modelling of a rudimentary sort, and well short of what would be needed to represent the structural intricacies of real software.

Deep learning changed the picture. Neural architectures brought a marked improvement in representation learning [23]; trained on large text and code corpora, they could pick out semantic regularities and so lent themselves to tasks such as code prediction and program synthesis [13].

The Transformer reshaped generative modelling once again. Self-attention lets a model relate elements of code structure to a natural language description even when the two sit far apart in the input [24]—a property whose value in software engineering is hard to overstate, since program logic routinely threads through many interdependent modules and functions [25].

Architecture, though, is only half the story; large-scale pre-training accounts for much of the recent progress [26]. Foundation models trained on heterogeneous data transfer surprisingly well across programming languages and task types [8], generalising past the particular environments and development contexts they were exposed to [27].

Code-specific foundation models have meanwhile become a research direction in their own right [5]. Trained on very large software repositories, they are meant to internalise the structure and syntax of programming languages [28], and on source code understanding and generation they outperform their general-purpose counterparts [23].

Viewed at a distance, the whole trajectory describes a move from rule-based systems to systems that learn from data [20]—and, with it, a growing capacity to model language, logic and structure jointly within a software engineering setting [1]. It is on this capacity that an AI-driven development ecosystem is beginning to take shape [4] (Figure 2).

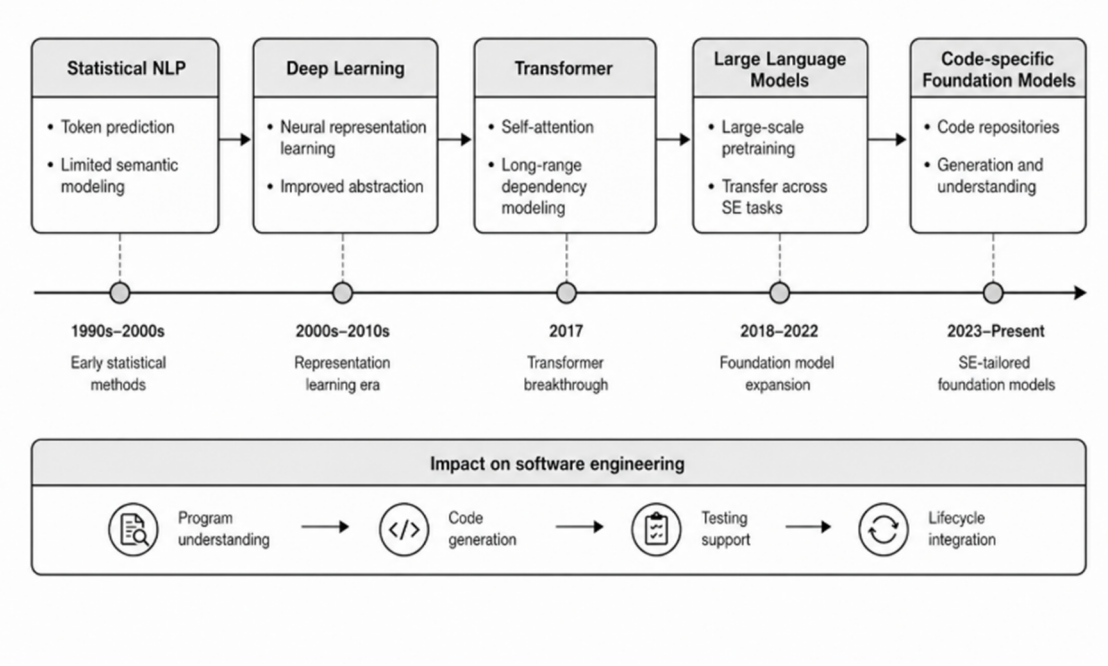


Figure 2. Evolution of Generative AI in Software Engineering.

3.2. Capabilities for Software Engineering Tasks

Several capabilities of generative AI bear directly on how software engineering is practised today. One of the most basic capabilities is to automatically generate code according to natural language descriptions [5]. A developer can state what is wanted in plain text and receive executable code in return [29], which trims the manual coding effort and, in turn, shortens development timelines [30].

Code completion is just as widespread [31]: given the surrounding context, the system predicts the fragment a developer is about to write. This not only improves development efficiency, but also reduces syntax and structural errors in the implementation process [32].

In addition, automatic test generation has become an important application field of generative artificial intelligence [7]. These models can generate unit tests, integrated tests and edge case scenarios according to the

program structure [9,16]. This helps to improve the test coverage and support the early detection of software defects [10].

Defect detection and program repair are also key competencies supported by generative models [18]. These systems can identify anomalies in the source code and propose repair patches [33]. In many cases, they can also provide explanations for detected problems to help developers understand and debug [34].

Requirements understanding is another emerging capability. Generative models can interpret natural-language descriptions and extract structured representations that support system design [3]. This can reduce ambiguity in requirements engineering and improve consistency during the early stages of development [35].

Beyond these core tasks lies support for specification work—turning informally stated needs into structured or semi-structured specifications that downstream tooling can act on [3]. Where user intent and system design tend to drift apart, this is precisely the gap such a capability helps to close [2].

In general, these capabilities show that the application of generative artificial intelligence is not limited to isolated tasks. On the contrary, it is increasingly applied to multiple interrelated stages of the life cycle of software engineering [1]. This creates a more integrated and intelligent development process, and is gradually reshaping the traditional software engineering practice [4].

3.3. Representative Models

Several representative models constitute the technical basis of generative artificial intelligence in the field of software engineering. The GPT series model represents a universal architecture, which excels in both natural language processing and code generation tasks [30]. These models can generate grammatically correct and context-related code based on text input [8].

Codex is a special model specially designed for programming tasks [5]. It is trained based on a large-scale code base and optimised for converting natural language descriptions to executable programs. Its development process proves the effectiveness of specific field training for software engineering applications [26].

CodeBERT introduces a different modelling strategy that focusses on the combined characterisation learning of code and natural language [23]. It is widely used in tasks such as code search and semantic alignment between documents and source code [28]. This enhances the system's ability to understand the relationship between the code and its function description.

StarCoder represents a new generation of basic models that focusses on code. It trains on diverse multilingual code data sets and supports multiple programming languages [17], thus showing stronger adaptability in heterogeneous software engineering environments [27].

Other emerging large-scale language models continue to expand the application ability of generative artificial intelligence in software engineering [13]. These models aim to improve robustness, reduce "hallucinations", and enhance consistency in multi-step programming tasks [19]. In addition, they also show better performance in complex reasoning scenarios involving software development [36].

In addition, recent empirical studies show that increasing the size of the model can usually improve its performance in software engineering benchmarks [27]. However, the improvement of ability does not always ensure the correctness or reliability of the results [37]. This highlights the importance of combining generative models with verification and confirmation mechanisms in the actual software engineering system [38].

4. Unified Framework of Generative AI in Software Engineering

Existing research on generative artificial intelligence often explores various software engineering tasks in isolation, and there is a lack of a unified framework that can run these tasks throughout the development life cycle [1]. At the same time, the agent-based method has begun to explore autonomous multi-stage workflow [20]. A broader systems perspective is also consistent with recent AI research that examines AI models and applications within an integrated system framework rather than as isolated technological components. This chapter therefore proposes a conceptual lifecycle framework that maps artificial intelligence capabilities to the stages of the life cycle and introduces a closed-loop model for organising generation, execution, testing, feedback and refinement. The model is intended as a synthesis of the reviewed evidence rather than as a validated replacement for established Agile, DevOps or MLOps processes.

4.1. Software Engineering Lifecycle Mapping

The life cycle of software engineering provides a structured perspective for system development activities. The traditional model describes the process as a series of clearly defined stages, including requirements analysis,

system design, coding, testing, deployment and maintenance. This structure has been widely discussed in software engineering research and is still a basic abstract concept [1].

Under the background of generative artificial intelligence, each stage of the life cycle can be reinterpreted from the perspective of intelligent automation [1]. In the requirements stage, generative models support the transformation of natural-language descriptions into structured specifications [3], thus reducing ambiguity and improving communication between stakeholders and developers [2].

In the design stage, generative artificial intelligence-assisted architecture suggestions and system decomposition. These models can analyse text requirements and propose structural design schemes [3], thus accelerating the speed of design exploration and supporting decision-making in the early stage of development.

In the coding stage, generative artificial intelligence plays a direct role in source code generation [5]. The model generates executable code based on natural language input and context prompts [29]. This ability significantly reduces the workload of manual realisation and shortens the development cycle [30].

In the testing stage, the generative model supports the automatic creation of test cases and the generation of test data [7], and assists in identifying boundary situations that are difficult to detect manually [10], thus improving software reliability and expanding test coverage [9].

In the deployment stage, generative artificial intelligence can assist in generating configuration information and realise deployment script automation, which helps to simplify the continuous integration and continuous delivery process in the modern software engineering assembly line [39].

In the maintenance stage, generative artificial intelligence supports defect repair, code reconstruction and system evolution tasks [33,40]. These capabilities enable software systems to continuously improve over time and reduce long-term maintenance costs.

Overall, life cycle mapping shows that the application of generative artificial intelligence is not limited to isolated tasks, but provides support for all stages of software engineering [13]. Compared with the traditional software engineering model, this builds a more unified and intelligent development process [4].

4.2. GenAI Capability Mapping

The generative artificial intelligence system shows a series of core capabilities, which can directly correspond to software engineering tasks. These capabilities constitute the cornerstone of artificial intelligence-driven software development systems and have been widely discussed in recent literature [6,17].

Natural language understanding is a basic ability to generate models, which enables the system to interpret user needs expressed in text form [3]. This ability is crucial to bridge the gap between human intent and software realization [2].

Code generation—program synthesis, in older terminology—is a second core competence: from natural language, or from a fragment of existing code, the model produces a complete program [5].

This is conducive to rapid prototyping and reduces the workload required for the initial implementation stage [29].

Reasoning ability enables the model to analyse program logic and identify the association between code components [36]. This enables the system to perform more complex software engineering tasks, such as defect analysis and system optimisation [24].

Support for Verification and Validation is an emerging capability in generative artificial intelligence systems. These models can generate test cases and assist in checking the correctness of the program [10,38]. Although the capability is still in the development stage, it plays an important role in improving software reliability and reducing defects.

These capabilities are combined to form a comprehensive system to support software engineering activities. They enable generative artificial intelligence to participate in multiple stages of development, not just as a single-purpose tool [13].

4.3. Closed-Loop AI-Driven SE Framework

The traditional software engineering process is often described as a linear assembly line, that is, following the sequential process from requirements analysis to system maintenance. However, this linear structure has limitations in adapting to the dynamic changes of the software system (Figure 3).

Generative artificial intelligence has contributed to the transformation from a linear assembly line to a closed-loop system [20]. Under this new paradigm, software development is regarded as a continuous circular process that includes generation, execution, feedback and optimisation [41]. Each iteration improves the quality of the software system through the adaptive learning mechanism.

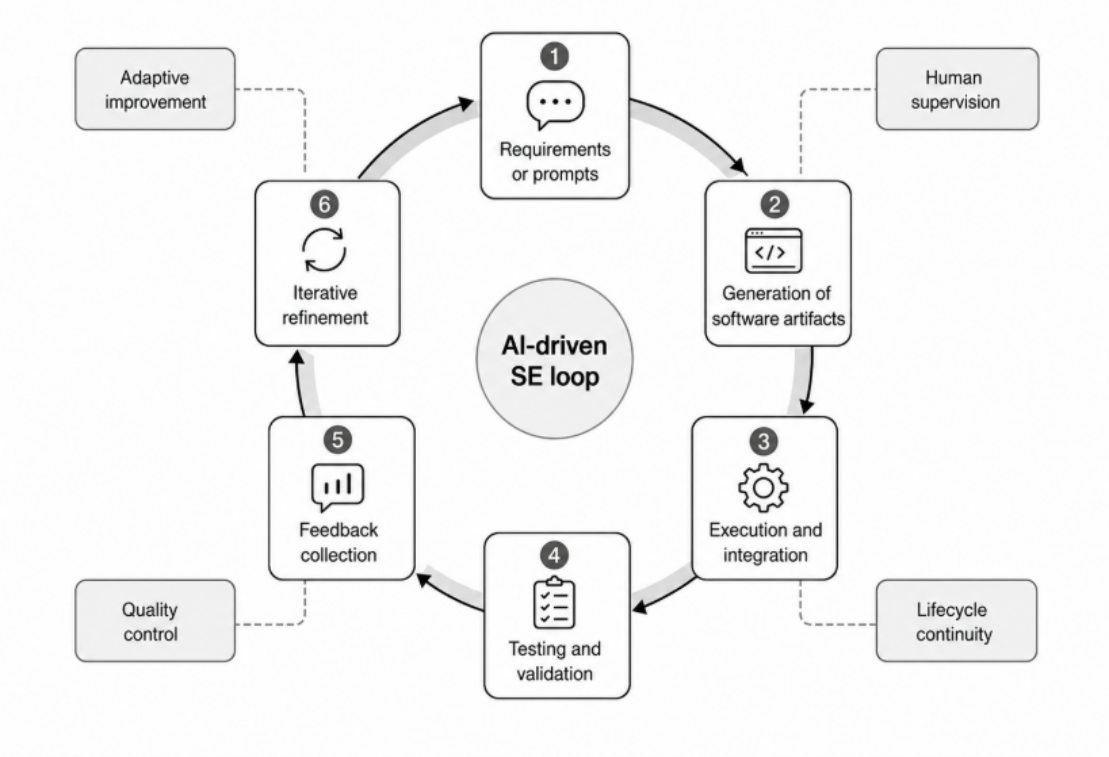


Figure 3. Closed-Loop AI-Driven Software Engineering Framework.

In the closed-loop framework, the generation model first generates code or design products according to the input specification [3]. Subsequently, these products are executed or tested in the system environment. The feedback signals generated by the execution results will be used to improve the subsequent generation process [42].

This continuous feedback cycle transforms software engineering into an iterative optimisation process. The system no longer only produces static output, but also evolves over time by generating repeated interactions between components and evaluating components.

Artificial intelligence-driven iterative optimisation plays a central role in the framework [43]. The generation model learns from the past output and adjusts the future generated content accordingly [44]. This helps to improve consistency, reduce errors, and improve the overall system performance in software development tasks.

From a more macroscopic perspective, this closed-loop model provides a conceptual abstraction of the shift toward more autonomous software engineering systems [20]. It summarises how generation, execution, verification and feedback may be linked while human supervision remains important. The model should therefore be read as an organising synthesis of recurring lifecycle interactions in the reviewed literature, not as evidence that a fully autonomous closed-loop process has already been established in practice [4].

5. Code Generation

Code generation has become the most widely researched application in the field of generative artificial intelligence. The relevant progress covers prompt engineering [45], parameter efficient fine-tuning [26] and multi-agent collaboration [43]. However, this field still faces many serious challenges, including code hallucination [19], insecure code patterns [46], and the lack of an evaluation framework other than functional correctness [6]. This chapter will analyse code generation methods based on prompts, context-aware generation and multilingual code generation, and discuss program repair techniques and challenges to be solved.

5.1. Prompt Based Code Generation

Code generation is one of the most mature and widely studied applications of generative artificial intelligence in the field of software engineering [5]. What it offers is a route from natural language description to executable code [47]—in effect, a direct mapping between what a person intends and what a machine can run. Recent work treats it as the basic bridge linking requirements specifications to their implementation [48].

The early literature confined itself largely to simple instruction-to-code settings: a short textual prompt goes in, the corresponding code comes out [45]. Success in this regime rests almost entirely on large-scale pre-training

and on pattern recognition over programming corpora, and it improves markedly when prompts are structured and spell out their functional constraints explicitly [49].

As the models matured, prompt-based programming moved past single-function generation. There is now evidence that they can sustain multi-step reasoning while constructing code [36]—which suggests learning that goes beyond surface grammar to higher-order program structure and logical dependency [24].

For practitioners the appeal is obvious: less hand-written code, shorter cycles [50]. The catch is that output quality remains hostage to prompt design and to the distribution of the training data [45]. Correctness, in other words, tracks the quality of the human input—a coupling that limits what the approach can deliver in practice [51].

5.2. Context Aware Code Generation

Context-aware generation takes the basic prompt mechanism and feeds it the surrounding code as well [52]. With that additional signal, the output can be made to respect a project's existing structure and its architectural constraints [53].

Analysis at the level of the whole code base—variable definitions, dependencies among functions, control flow [25]—gives such models a view that a lone prompt cannot supply. Code generated against this backdrop conforms more closely to system-level design requirements [54] and keeps the modules of a project in better agreement with one another.

One advance deserves particular mention: dependencies that stretch across file boundaries can now be captured [25]. Large systems, whose components are densely interconnected, benefit most. Technically, the capability rests on self-attention, which the Transformer architecture supplies.

Structural consistency, however, is not design understanding. Implicit intentions still escape these models [55]; many systems carry architectural assumptions that were never written down in the source [56], and inferring that unstated logic inside a complex code base remains an open difficulty.

5.3. Program Repair and Code Refactoring

Program repair extends generation into the corrective domain: the model is asked to find defects and put them right [57]. Its close bearing on maintenance and on reliability explains the volume of attention it has drawn [33].

In the usual arrangement the system inspects execution traces, error logs or the offending fragment itself [58], and from that evidence proposes a candidate patch for the observed fault [39]. Set against traditional workflows, the hours spent in manual debugging fall considerably.

Refactoring is repair's near neighbour, though its object differs: internal structure is improved while externally visible behaviour stays fixed [40]. Readability, modularity, reduced complexity [59]—gains of this kind matter little on any given day and a great deal over the life of a system, where maintainability and the capacity to evolve are decided.

Correctness and stability nonetheless remain genuine problems for repair systems [60]; a generated fix can itself introduce behaviour nobody asked for, or defects that stay hidden [61]. This limitation highlights the need to integrate verification or test-based verification mechanisms in the repair process [62].

5.4. Multi Language Code Generation

Modern generative AI systems increasingly support the processing of multiple programming languages in a unified architecture [8]. This capability is due to large-scale multilingual training data sets containing multiple programming languages and frameworks [17].

The multilingual code generation function allows the model to convert or generate code with equivalent functions between different programming languages [63]. This is especially important in heterogeneous software systems with components realised by different technologies, and also supports legacy system migration and cross-platform development tasks [64].

In addition, this capability also improves the convenience of developers working across different programming ecosystems [65]. A single unified model can support multiple programming languages without retraining for specific languages [54], which significantly enhances the flexibility of deployment in the actual development environment.

However, the characteristics of a specific language are still a major challenge to realise model generalisation [31]. Various programming languages are different in syntax, semantics and runtime behaviour [66]. These differences may reduce the accuracy of generation in complex translation or multilingual integrated tasks [63] (see Table 2).

Table 2. Summary of Code Generation Approaches.

Approach	Description	Key Advantage	Key Challenge
Prompt-based	NL instruction to code translation	Reduces manual coding effort	Sensitive to prompt quality
Context-aware	Incorporates project-level code context	Structural consistency	Difficulty with implicit design logic
Program repair	Identifies and patches code defects	Accelerates debugging	May introduce new defects
Code refactoring	Improves internal structure without changing behavior	Enhances maintainability	Limited domain-specific understanding
Multi-language	Cross-language generation and translation	Supports heterogeneous systems	Language-specific semantic gaps

5.5. Challenges in Code Generation

Although code generation technology based on generative AI has made significant progress, there are still some fundamental challenges that have not been solved. One of the most critical problems is "code hallucination", that is, although the code generated by the model is grammatically correct, it is functionally wrong [19]. Reports of the phenomenon recur across models and across benchmarks [18].

A second limitation is the absence of any formal guarantee of correctness [38]. Code that clears the basic test cases can still break at boundary conditions or under production workloads [51]—an uncomfortable prospect wherever software is safety- or mission-critical [67].

Security adds a further layer of concern [46]. Insecure idioms—careless data handling, misused APIs—find their way into generated code [61,68], and the recent literature argues for security-aware training and verification as the countermeasure [69,70].

Evaluation, finally, is unfinished business [6]. Pass rates and similarity-based measures capture neither semantic correctness nor functional adequacy in full [14], so comparisons between generative models across studies rest on shaky ground [47].

6. Software Testing and Debugging

In testing, large language models have shown promise on several fronts at once: unit tests with high coverage [16], defect reproduction from natural language reports [34], and mutation-based techniques that sharpen test effectiveness [10]. However, challenges such as test oracle problems, non-deterministic behavior [37] and scalability limitations are still serious. This chapter will discuss automated test generation, fuzz testing, fault localization, self-healing systems, and the principal barriers to practical deployment.

6.1. Automated Test Generation

Automated test generation is a basic application of generative artificial intelligence in the field of software engineering [71]. It transforms software testing from an artificial design activity to a model-driven generation process [7] (Figure 4). Recent research shows that large language models can generate unit tests and integrated tests directly according to the source code and function description [72].

These models acquire knowledge by learning the test mode in the large-scale software code base [73] and capture the relationship between the input conditions and the expected program output [74]. This enables them to generate test cases that cover both the normal execution path and contain boundary conditions [10].

In addition to basic unit tests, generative artificial intelligence also supports the generation of system-level and functional-level tests [75]. This makes it possible to verify complex software behaviour under different runtime conditions, which is especially important for large-scale systems with incomplete manual testing designs or high costs [71].

More recent work pairs test-suite generation with retrieval-augmented generation (RAG) [76]. Suites built this way adapt to the input space at hand rather than being fixed in advance, which makes verification more flexible and extends functional coverage [77].

Effectiveness, even so, is bounded by prompt quality and by how diverse the underlying data happen to be [16]. Generated tests frequently arrive without meaningful assertions [78]—a reminder that human guidance still stands between a test and its semantic correctness [79].

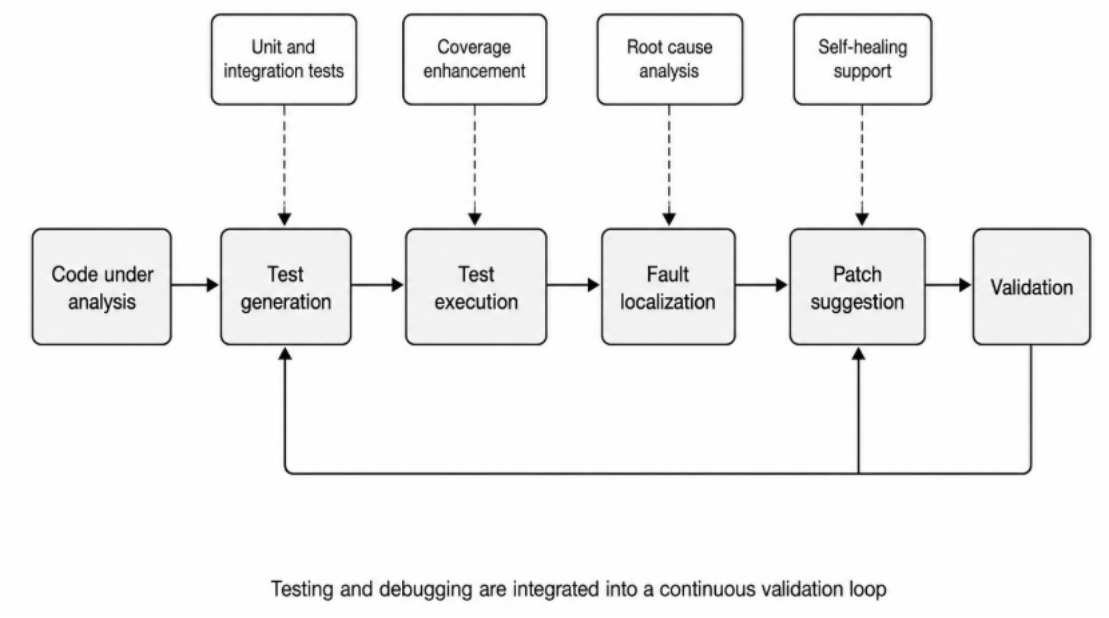


Figure 4. AI-Driven Testing and Debugging Pipeline.

6.2. Test Optimization and Input Generation

Test optimisation is concerned with making the testing process cheaper and faster [80]. Here the contribution of generative AI is chiefly analytical: it reads the existing suite and points to what remains uncovered. Branch and path coverage both improve measurably under AI-based methods, according to recent studies [81].

For paths that no test exercises, such systems suggest additional inputs—some mined directly from defect reports [82]. Blind spots in verification shrink as a result, and the software stands up better to inputs nobody anticipated [83].

Regression testing is a second target. By ranking test cases against recent code changes and execution history [80], a generative model keeps fault detection high while holding compute costs down.

Minimisation follows naturally: redundant cases are pruned without giving up coverage, which matters in continuous integration settings where the suite runs constantly [84]. What has proved harder is settling on the right coverage criterion in the first place. A generated test can raise the coverage figure without raising the odds of catching a fault [81], and in that gap structural coverage and functional correctness part company.

6.3. Fault Localization

Automated debugging occupies a central place in AI-driven software engineering research [85]. The task is twofold—trace a failure to its root cause, then help the developer repair it—and deep learning approaches, small language models among them, have been shown to sharpen localisation accuracy [86].

In operation, the system reads execution traces and error logs, flags the fragments that look suspect [85], and ties the failure mode back to the program's structure [58]. Debugging takes less time; developers get more done.

Explanation is a useful extension [34]: the probable cause of a failure is described in natural language, which makes the diagnosis easier to interpret and the system's behaviour easier to grasp.

A few studies push further, into multi-step debugging reasoning [33]. The model there imitates how a human developer would work through a problem, and the fault analysis it yields has more structure than anything single-step prediction produces.

Complexity is where the approach runs out of road. When many components interact, as they do in any large system, root cause analysis becomes genuinely hard [58], and accuracy in real debugging scenarios suffers accordingly.

6.4. Self-Healing Systems

Self-healing systems mark the far end of automation in software maintenance. Running software is watched continuously and anomalies are flagged without human involvement [87]; automated logging supplies the record from which behaviour is understood and faults are diagnosed.

On failure, the generative model attempts a patch [57], drawing on repair patterns learned from historical data [33]. Sometimes—not always—recovery completes with no one intervening at all.

The usual deployment couples the mechanism to the continuous integration pipeline [22], so that errors introduced as the software evolves are caught and corrected at once [39]. Stability improves; in large applications, so does the maintenance bill.

Iterative self-repair goes a step further [88]: patches are generated, tested, discarded and regenerated until one holds [42], closing a feedback loop between the generating and the verifying sides of the system.

Full autonomy, though, is not yet within reach [18]. A patch can bring inconsistencies of its own [60], which is reason enough to keep verification at the heart of any automated repair system [38] (Table 3).

Table 3. Summary of Testing and Debugging Approaches.

Approach	Description	Key Advantage	Key Challenge
Automated test generation	Generate unit/functional tests from code or specs	Improved coverage	May lack meaningful assertions
Fuzz testing & input generation	Generate diverse test inputs for untested paths	Enhanced robustness	Difficulty defining coverage criteria
Fault localization	Identify root causes of failures	Reduced debugging time	Limited accuracy in complex systems
Self-healing systems	Automatic patch generation and iterative repair	Enables autonomous recovery	May introduce new inconsistencies

6.5. Challenges in Testing and Debugging

For all the progress, AI-driven testing and debugging carries unresolved difficulties, and the oracle problem heads the list: for a test case produced automatically, deciding what the correct output ought to be is often no easier than the original testing task, and the accuracy of evaluation is capped as a result [6].

Another challenge is the non-deterministic behaviour of the generative model [37]. The same input may produce different outputs in multiple runs [89], which reduces the reproducibility of the testing workflow.

Scalability is also a major limitation. Large-scale software systems require extensive analysis of code and execution trajectory, and existing models are often difficult to maintain good performance in complex and large-scale environments [90].

In addition, there is a gap between structural correctness and functional correctness. The generated test cases may achieve high code coverage, but cannot detect actual software failures [10]. This shows that coverage indicators alone are not enough to evaluate the quality of the test [14].

Another challenge is how to integrate it into the actual development pipeline [84]. Many generative test systems run in an isolated environment [91], which limits their uptake in industrial-grade continuous integration.

7. Human–AI Collaboration in Software Engineering

Generative AI is promoting changes in software engineering: from a human-centred discipline to a collaborative paradigm, that is, developers and AI systems work together to develop software artifacts [2,92]. This transformation redefines the role of developers, reshapes the organisational structure, and triggers fundamental thinking about trust and control in artificial intelligence enhancement workflows [15]. This chapter will discuss this change from five dimensions: human-in-the-loop programming, AI-assisted paradigm, collaboration mode, impact on developers, and trust and control issues.

7.1. Human-in-the-Loop Programming

Human-in-the-loop programming represents a fundamental change in software engineering under the generative AI system [64]. Traditional programming assumes that developers are fully responsible for the design, implementation and verification of software logic in a deterministic workflow. In contrast, generative AI introduces a probabilistic and interactive development environment, and the code is jointly constructed by humans and intelligent systems [93].

The interaction is better described as a standing feedback loop between human intention and machine generation than as a handover of tasks. A developer states a need in a prompt; the system answers with candidate solutions [50]; and those candidates are then judged, refined and reworked over several cycles [94].

Cognitively, the paradigm redistributes where programming knowledge is applied [56]. Attention moves up a level—away from syntax, toward semantics: decomposing the problem, pinning down constraints, judging the

resulting program. Programming, on this account, is becoming intention-driven where it used to be implementation-driven.

Uncertainty enters as well. The same input can yield several different outputs [37]—something deterministic programming never had to contend with—so developers find themselves weighing alternatives against one another, and the decision structure of the task shifts under them.

On routine tasks the productivity gains are empirically documented [93]. They are not automatic: how much the interaction delivers depends on the developer's expertise and on whether the prompts are precise. Skill in steering the generative system, in short, still matters [45].

7.2. AI-Assisted Programming Paradigms

Of all the ways generative AI has entered software engineering, AI-assisted programming is the most visible [11]. Embedded in the development environment, these systems offer code suggestions in real time along with broader support, and the day-to-day workflow of development has been reorganised around them [92].

The AI functions here as a persistent layer of cognitive support: from partial input and context it anticipates the code structure to come [32], relieving some of the burden of repetitive coding and quickening the pace of development [95].

A key feature of this paradigm is adaptive context learning. The model will adjust its output according to the project structure, coding specifications and historical code mode [96], which not only improves the integration with the existing software system, but also enhances the consistency of large-scale code libraries.

However, over-reliance on AI assistance can also bring behavioural risks [56]. Developers may gradually reduce their participation in underlying coding tasks, which may weaken their deep understanding of the internal mechanism of the system and reduce their debugging ability over time [97].

7.3. Human AI Collaboration Models

Human-computer collaboration in software engineering can be divided into multiple modes according to the different distribution of autonomy [92]. These models reflect the different degrees of control sharing between developers and generative systems (see Figure 5).

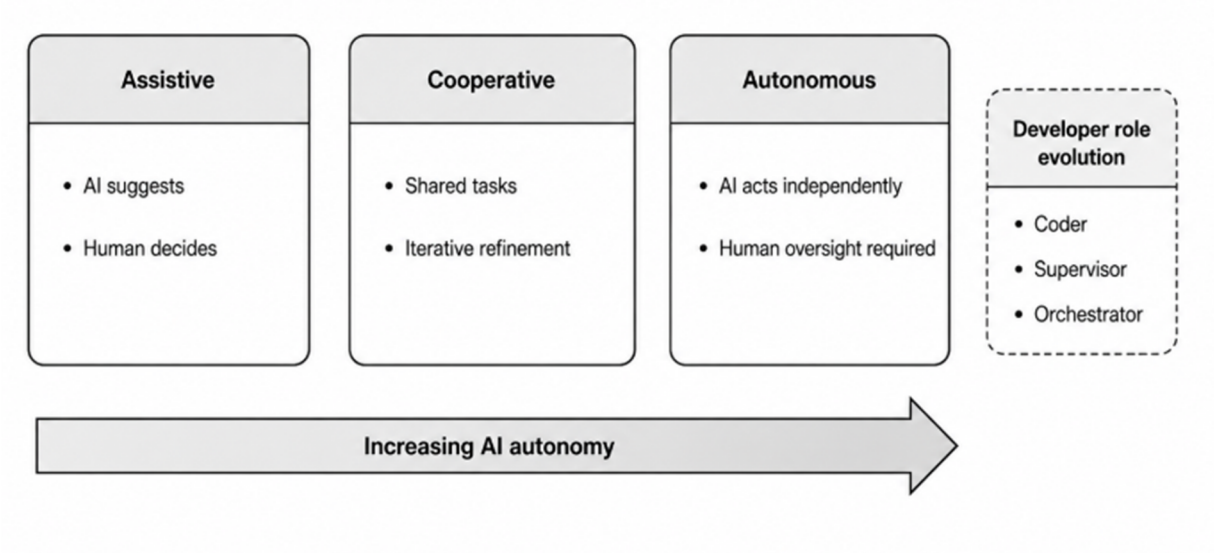


Figure 5. Human-AI Interaction Models in Software Engineering.

Related HCI research shows that user attributes and structural information can support user modelling, highlighting the importance of accounting for user heterogeneity in human–AI collaboration [98]. The Assistive model represents the lowest degree of autonomy. In this mode, AI provides advice, and developers retain full control over decision-making [11]. This model is widely used in modern integrated development environment (IDE).

The cooperative model represents a structure of joint decision-making [2]. Humans and AI jointly participate in the construction and improvement of software [41], and task allocation is dynamically adjusted according to the complexity of the system and the input of developers.

The Autonomous model represents the highest degree of independence of AI [20]. In this mode, the AI system generates and modifies software artefacts with very little manual intervention [99]. Although this improves the level of automation, it also brings risks in terms of reliability, responsibility attribution and control [100].

These models show that software engineering systems are undergoing a gradual evolution from human-centred programming to AI-enhanced and some autonomous systems.

7.4. Impact on Developers

Generative artificial intelligence has greatly reshaped the cognitive structure of software development. For the individual, hours once spent typing code are being reallocated to reasoning, verification and system-level design [94]—execution giving way, in effect, to decision-making.

The cognitive workload is redistributed along with the hours. Interpreting what the model produces, judging whether it is correct, folding it into a larger system—these responsibilities now fall to the developer [94]. This requires stronger analytical ability and architectural thinking ability.

At the team level, the software development process is deeply integrated with artificial intelligence. Generation models are increasingly embedded in the development process and dialogue interface, acting as collaborative intelligence [101]. This change changes the task allocation pattern within the team and reduces the importance of repeated programming roles.

In addition, the role of software engineer is also constantly evolving. Developers are gradually changing from code writers to system coordinators [4]. This role focuses on supervision, coordination and verification, rather than direct code implementation.

Long-term dependence on generative artificial intelligence may affect the development of skills [56]. Failure to balance with traditional learning practices, reducing exposure to underlying coding tasks may weaken basic programming skills [97]. This has an important inspiration for software engineering education and practitioner training.

The effect on junior developers is likely to be mixed rather than uniformly positive or negative. AI assistance can lower the barrier to completing routine tasks and expose less-experienced developers to examples and explanations [93], but excessive substitution of foundational coding practice may weaken code comprehension and debugging skills over time [56,97]. Software engineering education should therefore combine AI-supported development with deliberate practice in programming fundamentals, testing and code reading. As routine code production becomes faster and less costly, greater value is likely to shift toward requirements specification, architectural reasoning, verification, security review, system integration and the ability to judge when AI output should be accepted, revised or rejected. This shift is consistent with the broader human-centric innovation perspective, which emphasises that technological transformation should augment human capabilities rather than simply replace them. Related Industry 5.0 research likewise identifies upskilling, reskilling, and the redesign of human roles as central requirements for effective human-AI integration [102].

7.5. Trust and Control Issues

Trust is the core element of the human-computer collaboration system [15]. Developers must constantly decide whether to accept or reject the output generated by artificial intelligence [94]. This decision-making is influenced by experience, system reliability and task complexity [92].

Excessive trust in generative artificial intelligence may lead to unnoticed software defects [60]. At the same time, insufficient trust will reduce the efficiency advantage brought by artificial intelligence assistance [15]. Therefore, in the collaborative development environment, it is necessary to establish a balanced trust calibration mechanism.

In software engineering systems, control and transparency are also crucial [100]. Developers need to understand how generative models generate outputs. Without interpretability, it is difficult to verify correctness and ensure system reliability in key applications [15].

Explainability plays an increasingly important role in improving the quality of collaboration. When the model provides reasoning instructions for the generated code, developers can better assess its correctness [103] and effectively integrate the output into the software system.

8. Software Quality Assurance

The code generated by the large language model (LLM) often has defects in maintainability, performance and security, which far go beyond the category of functional correctness [12]; at the same time, the non-deterministic generation process also complicates the traditional quality assurance process [37]. Existing research shows that there is a significant imbalance: compared with other quality dimensions, functional correctness has

received more attention [6]. This chapter will discuss the quality dimension, artificial intelligence-based code review, vulnerability detection, the risks of generative artificial intelligence and verification methods.

8.1. Software Quality Dimensions

In a software engineering system driven by generative artificial intelligence, software quality assurance is a basic consideration [12]. Traditional software engineering defines quality by attributes such as reliability, maintainability, security and efficiency [6]. These attributes are still applicable in artificial intelligence-generated software systems. However, due to the probabilistic and non-deterministic characteristics of the generative model, its evaluation becomes more complicated [37] (Figure 6).

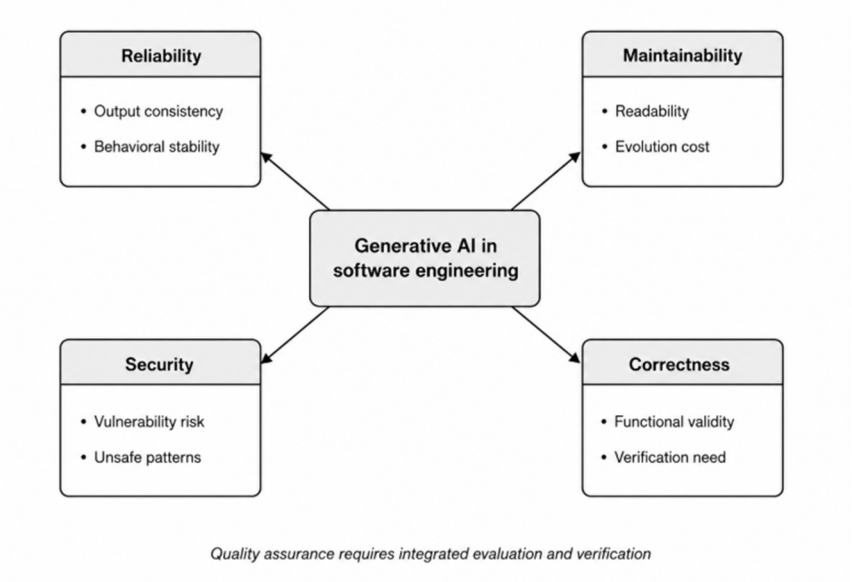


Figure 6. Impact of Generative AI on Software Quality Dimensions.

Reliability in generative artificial intelligence system refers to the stability of output under similar input conditions [51]. Empirical research shows that the same prompt may produce different code outputs in multiple executions [37]. This brings variability to software behaviour and reduces the predictability of the development workflow [89].

Maintainability is also significantly affected by generative artificial intelligence [12]. Although the code generated by artificial intelligence can accelerate the development process, it may also introduce less modular or difficult-to-understand code structures [104]. Over the long run the system becomes harder to maintain, and evolving it costs more [56].

Security belongs on the list as well [46]. Insecure patterns slip into generated code without anyone intending them [68], and unless extra analysis tooling or a verification framework is in place, they tend to surface late rather than early [105].

Efficiency has begun to attract attention of its own [106]. LLM-generated code, the evidence suggests, can underperform hand-written code—consuming more energy, occupying more resources.

Correctness, finally, carries growing weight in AI-generated systems [38]. Structural checks that sufficed under deterministic code generation no longer guarantee it [62]; assessment has to reach down to the semantic level [107].

8.2. AI-Based Code Review

The code review system based on artificial intelligence has become an important application of generative artificial intelligence in the field of software engineering [96]. These systems detect potential defects or style inconsistencies by analysing the source code. The generative artificial intelligence model can provide semantic-level feedback beyond the level of syntax analysis [107], so as to identify deeper design problems in the software system.

In addition, artificial intelligence systems can also make reconstruction suggestions during code review [40]. These suggestions help to improve code readability and reduce technical debt during long-term software maintenance [12].

However, the automated code review system still has limitations in understanding the logic of specific fields [108]. This may lead to false alarms or missed inspection defects in complex systems [105].

8.3. Security and Vulnerability Detection

Security analysis is the core component of software quality assurance in generative artificial intelligence systems. Recent research shows that generative models can be used to detect unsafe coding patterns in a variety of programming languages [105]. These systems identify potential vulnerabilities by analysing the source code, such as unsafe API use or unsafe data flow [63]. Risks are thereby caught early in the development pipeline rather than after release [39].

The same technology, though, opens new avenues of risk. Chief among them: the system may itself emit insecure code [69]. A model that learns from massive corpora will reproduce whatever unsafe patterns those corpora contain [70].

Adversarial attacks on code-generating models are a newer worry [109]. By manipulating the prompt, an attacker can bend model behaviour in ways the operator never anticipated [109], planting vulnerabilities inside automated generation and deployment chains.

Dependency hallucination deserves separate mention [19]. A model that cites a package which does not exist—or one that is unsafe—exposes the real software supply chain to serious risk [100].

Taken together, these problems make one point plain: securing generative AI systems means preventing flaws at generation time, not merely detecting them afterwards [46].

The question is therefore not whether AI-generated code is inherently more or less secure than human-written code. Current evidence does not justify a general claim of systematic superiority: security can improve when models are guided by secure coding constraints, security-aware prompting, contrastive decoding or dedicated verification [46,68–70], but generated code can still reproduce unsafe APIs and patterns from training data [61]. For autonomous coding agents, mitigation requires layered controls that combine static and dynamic analysis, dependency and vulnerability scanning, sandboxed execution, test-based validation, provenance logging and human approval at high-risk merge or deployment points [39,105,109].

At scale, a further concern is correlated risk. If the same models, prompts or suggested dependencies are reused across many systems, one insecure pattern or hallucinated dependency may be replicated widely rather than remaining a local defect [19,61]. This creates a software-supply-chain concern in which diversity of tools, continuous dependency monitoring and rapid patch propagation become as important as the security of any individual generated component.

8.4. Verification and Validation

Verification and validation have always been load-bearing elements of quality assurance, traditionally carried out through structured testing, formal methods and manual review. Where generative AI is involved, the same processes are being folded, step by step, into automated pipelines [22].

One contribution the models make is producing verification test cases against which their own output can be checked for functional compliance [10]; divergence between specification and implementation then shows up early rather than late.

Hybrid schemes that pair formal verification with generative models are under active exploration [99]; symbolic reasoning is coupled to neural components [110], and the pairing strengthens both the assurance of correctness and the reliability of the whole.

Comprehensive formal verification of AI-generated code, however, has not been achieved. Current models operate without strict logical consistency constraints, and no guarantee of correctness across every execution scenario can be given—a limitation that keeps them out of safety-related domains and autonomous systems [67,111].

Verification practice must also come to terms with software that is probabilistic in origin: rather than checking against a single expected result, as deterministic systems allow, it has to reckon with a set of possible outputs [37].

8.5. Evaluation Metrics and Frameworks

The absence of full formal verification bears directly on how evaluation should be framed. Because current models enforce no strict logical consistency, correctness across every execution scenario cannot be certified, and metrics that quietly presuppose deterministic behaviour lose part of their meaning in safety-related and autonomous settings [67,111].

Any credible evaluation framework must likewise accommodate probabilistic output: a single expected result is no longer the right yardstick when one and the same request can legitimately produce several answers [37].

Prevailing indicators, moreover, weight structural attributes over semantic correctness [14]. This leads to a gap between the measured performance and the reliability of real-world software.

Therefore, future research should focus on developing a comprehensive evaluation framework that integrates correctness, reliability, security and maintainability into a unified evaluation model [6,12].

9. Discussion

The previous chapters show that generative AI is promoting a paradigm shift: from a certain and human-centred workflow to a probabilistic, AI-enhanced engineering system [1,4]. However, there are still urgent challenges to be solved in terms of correctness [19], evaluation standardisation [14] and the long-term impact on developer skills [56]. This chapter will synthesise the key findings, summarise the proposed classification system, evaluate its industrial impact, and outline the future research direction.

9.1. Paradigm Shift in Software Engineering

Generative AI is triggering the fundamental restructuring of the discipline of software engineering [1]. Traditional software engineering is based on a deterministic workflow in which developers clearly define system logic and execution rules. In contrast, generative AI has contributed to a probability-driven and data-driven paradigm, under which software artefacts are generated through learned characterisation, not through manually coded instructions [4] (Figure 7).

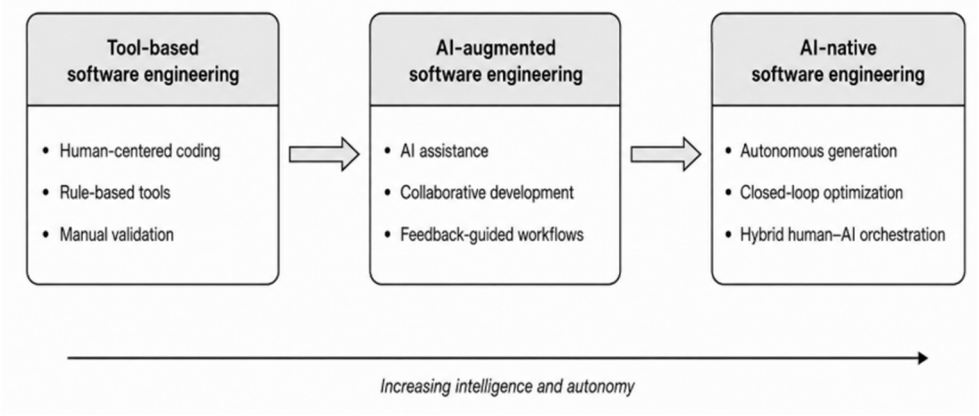


Figure 7. Paradigm Shift in Software Engineering.

This change has led to a structural change in the essence of software engineering work. The discipline no longer takes manual implementation as the core, but transforms into an interactive process with generative systems, which are responsible for generating, refining and optimising software artefacts [94]. The role of developers is increasingly changing from simple implementers to supervisors and evaluators [56].

Under this new paradigm, reasoning ability has become more and more important. Developers need to define high-level constraints and interpret system-level output, rather than just focussing on coding at the grammar level [35]. This marks the transformation of the engineering model from realisation-centred to standard-centred and intention-driven [2].

At the system level, software development becomes more dynamic. The generative model will continuously update the output results according to the feedback of the test and execution environment [42]. This model brings an adaptive and evolving software engineering process, which is completely different from the traditional static development pipeline [20].

9.2. Addressing the Research Questions

This subsection revisits the five research questions and summarizes the principal findings from Chapters 4 through 8.

RQ1. Unified lifecycle integration.

The analysis of Chapter 4 shows that generative artificial intelligence provides functional support for all stages of the life cycle of software engineering. In the requirements stage, the generative model transforms natural language into structured specifications; in the design stage, they assist in providing architectural suggestions; the coding link benefits from direct source code generation; the test link benefits from automated test case generation

and boundary identification; in the deployment and maintenance stage, it automates and iteration through configuration. Defect repair is enhanced.

The study further clarifies the four core artificial intelligence capabilities that support the above applications: natural language understanding, code generation, logical reasoning and verification support. These abilities are not isolated collections of functions, but constitute an interrelated layer of capabilities. In addition, the closed-loop framework proposed in this article transforms the linear development process into a continuous cycle including generation, execution, feedback and optimisation links, so as to achieve iterative improvement through the adaptive learning mechanism. This closed-loop representation is used as an organising lens rather than as an externally validated process model.

RQ2. Code generation approaches and challenges.

Chapter 5 explains the five main methods. Prompt-based generation converts natural language into code, and its ability has evolved from simple function synthesis to support multi-step reasoning. Context-aware generation uses information at the code base level to ensure that the generated content is consistent with the existing project structure. Program repair aims to identify defects and provide corresponding repair patches. Code reconstruction optimises the internal structure without changing external functions. Multilingual generation supports cross-language conversion within a unified architecture.

There are still some unresolved challenges. Code hallucination can result in syntactically valid but functionally incorrect generated code. The lack of formal correctness guarantee limits its application in security-critical systems. When the model reproduces the unsafe mode in the training data, it will cause security vulnerabilities. The existing assessment indicators cannot comprehensively measure semantic correctness. These limitations show that code generation technology is not mature enough to be deployed without manual intervention.

RQ3. Testing and debugging contributions.

Chapter 6 explains how generative artificial intelligence extends automated testing beyond traditional methods. Large language models can generate unit tests and functional tests according to the source code and specifications; they learn test patterns from the code base and cover standard paths and boundary conditions. The advanced method also combines retrieval enhancement technology to generate an adaptive test suite.

Generative models also play a role in prioritising test cases, simplifying test sets and fault localization. Some systems can generate natural language interpretations for detected defects, thus enhancing interpretability. The self-healing system further expands these capabilities by continuously monitoring the execution process and using iterative cycles to generate repair patches. However, test oracle problems, non-deterministic behavior, and scalability challenges in large-scale systems are still major obstacles to be overcome urgently.

RQ4. Human AI collaboration and developer roles.

Chapter 7 points out a fundamental change in software development practice. Human-in-the-loop programming establishes a feedback mechanism: developers express some requirements through prompts and iteratively optimise the solutions generated by AI. This model transforms programming from a deterministic implementation process to a probabilistic collaborative construction process; the focus of developers is also increasingly shifting to semantic reasoning and program evaluation, rather than code writing at the grammar level.

The study summarises human-computer collaboration into three models: the assistive model retains the full control of developers; the cooperative model allocates decision-making power between humans and AI; and the autonomous model grants the AI independence, with only minimal human supervision. At the individual level, developers are changing from simple code producers to system arrangers; at the organisational level, workflows are becoming more closely integrated with AI systems. In this process, trust becomes a key factor: excessive trust may lead to undetected defects, and insufficient trust will weaken the effectiveness of efficiency improvement. This highlights the need to ensure transparency and interpretability in human-computer collaboration.

RQ5. Quality assurance concerns.

Chapter 8 points out multiple quality dimensions affected by generative artificial intelligence. Probabilistic output means correctness can never be promised outright; non-determinism eats into reliability; maintainability suffers wherever the generated code is hard to read; unsafe patterns inherited from training data translate into security exposure; and performance regressions in AI-written code raise questions about efficiency as well.

Verification today leans on pass@k, code coverage, defect detection accuracy and similar measures—useful reference points, none of which adds up to a complete evaluation. Recent studies have explored hybrid approaches that combine formal verification with neural methods to strengthen correctness assurance [99,110]. However, complete formal verification remains an open challenge. Analysis shows that there is a significant gap between

the current assessment practice and the comprehensive evaluation required for production deployment, which highlights the need to establish a unified framework that can comprehensively consider correctness, reliability, safety, maintainability and efficiency.

9.3. Taxonomy Summary

For the purposes of this review, the literature is organised into a four-dimension analytical taxonomy covering code generation, testing and debugging, human–AI collaboration and software quality assurance [13]. The taxonomy integrates the reviewed evidence into a common structure for comparison across the corpus [6] (see Table 4). It is intended as an organising device for synthesis rather than as a universally validated classification.

Table 4. Unified Taxonomy of Generative AI in Software Engineering.

Dimension	Sub-Categories	Focus
Code Generation	Prompt-based; context-aware; program repair; multi-language	Automation and efficiency
Testing & Debugging	Test generation; fuzz testing; fault localization; self-healing	Reliability and defect reduction
Human–AI Collaboration	Human-in-the-loop; AI-assisted; cooperative; autonomous	Workflow and role transformation
Quality Assurance	Vulnerability detection; code review; verification	Correctness, security, robustness

The first dimension of the classification system is code generation. This covers tasks such as natural language code conversion, program synthesis and code completion [5]. Relevant research focusses on improving the efficiency of automation and reducing the workload of manual programming [29].

The second dimension is testing and debugging. This includes automated test generation, fault localization, and self-healing systems [7]. These methods aim to improve software reliability and reduce the spread of defects between stages of development [71].

The third dimension is human-computer collaboration. This includes human-in-the-loop programming, collaborative development mode and artificial intelligence-assisted coding environment [92]. Relevant research focusses on changes in developer cognition, workflow organisation and decision-making process [56].

The fourth dimension is software quality assurance. This includes vulnerability detection, automated code review and verification support system [12]. These studies are dedicated to solving key issues related to correctness, security and system robustness [46].

In addition, the multimodal method of code search and synthesis by combining code examples and natural language descriptions is becoming an emerging cross-field research topic [112].

In general, the classification system comprehensively organises the application of generative artificial intelligence in software engineering. It not only facilitates cross-research comparison, but also highlights the integration trend of different research directions under a unified framework.

The contribution of the present review is therefore integrative rather than exclusive. Table 5 shows how its lifecycle-oriented scope differs from representative recent surveys that concentrate on a single task family, a broad set of LLM tasks, or agentic workflows.

Table 5. Comparison with Representative Recent Surveys.

Review	Primary Scope	Coverage	Distinctive Emphasis
Jiang et al. [5]	LLM code generation	Coding-centric	Methods and evaluation for code generation
Zheng et al. [13]	LLMs across SE tasks	Multi-task	Broad task-level understanding of LLM use in SE
Faraji & Pombo [7]	AI-driven test automation	Testing-focused	Testing techniques, tools and challenges
Otoum & Elkhalili [20]	Agentic software engineering	Multi-stage	Agent methods and autonomous workflows
Present review	Code generation; testing/debugging; human–AI collaboration; quality assurance	Lifecycle-oriented	Cross-stage synthesis linking technical, human, security and QA issues

To complement the conceptual figures with a data-driven view of the corpus, Figure 8 reports the publication-year distribution of the final included corpus, rather than the full search eligibility window. All 113 retained studies were published between 2024 and 2026, with 97 appearing in 2025–2026.

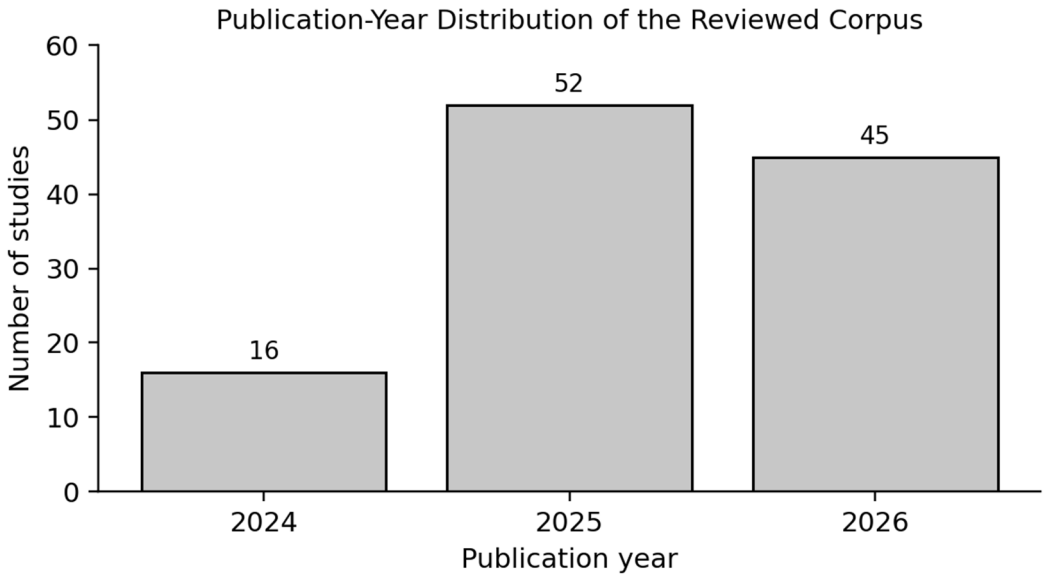


Figure 8. Publication-Year Distribution of the Reviewed Corpus.

9.4. Industrial Impact

Generative artificial intelligence has had a significant impact on industrial software engineering practises. AI-assisted development systems such as GitHub Copilot have been widely integrated into the modern programming environment [11].

These systems improve development efficiency by providing real-time code suggestions, debugging assistance and document support [32], thus shortening development time and improving the productivity of personal and team workflows [50].

In addition, organisations are increasingly integrating generative AI into the DevOps pipeline [22]. Automated testing, continuous deployment and system monitoring all ride on that integration [39], and software delivery becomes correspondingly more continuous and less manual. Generative AI adoption extends beyond task automation to reshape workflows, human roles, decision-making, responsibility allocation, and governance [92,98].

Industrial practice has also exposed the technology's rough edges [92]. Code reliability, security vulnerabilities and the danger of leaning too hard on the AI remain live concerns [15]—enough to argue for controlled rollout, and for keeping human supervision in the loop wherever these systems are deployed [100].

9.5. Ethical Considerations

Wide adoption has brought ethical questions along with the technical ones—questions of intellectual property, of data privacy, of environmental cost, and of responsible use in the classroom.

Ownership and attribution come first. Models trained on vast open-source corpora can emit output closely resembling copyrighted material, and with that resemblance come questions of legal liability and provenance. Watermarking and attribution mechanisms for tracing AI-generated code have been studied [100], but the trade-off between robustness and deployability means reliable source tracking remains some distance away.

Privacy runs a close second. The corpora needed to train large models are big enough to sweep in proprietary code, and once such a model is deployed in industry its output can leak confidential material. Collaboration on open source across organisational lines only multiplies the exposure [91].

Environmental sustainability is drawing scrutiny too. Training and running large models is computationally expensive, and comparative work finds wide variation in energy use and carbon output across models and tasks [106]—variation large enough that ecological impact belongs among the criteria for model selection.

Education presents a dilemma of its own. With AI tools now part of the curriculum, the worry is that students will lean on them until basic programming skill atrophies [97]; curricula will need deliberate design if the tools are to aid learning rather than substitute for it.

None of these questions will be settled by any single constituency; researchers, practitioners and policymakers will have to answer them together. The task ahead is a governance framework under which productivity gains can coexist with the protection of intellectual property, of privacy, and of integrity in education.

9.6. Key Challenges

Rapid as the progress has been, several foundational problems remain open. Correctness leads the list: code can be syntactically flawless and still fail once it meets real operating conditions or a sufficiently complex environment [19,51].

Evaluation criteria have yet to converge. Benchmarks of the pass@k or coverage variety measure neither semantic correctness nor system-level reliability in full [14], and results reported by different studies consequently resist comparison [6].

Scale is a bottleneck of its own. Models that acquit themselves well on small generation tasks falter when confronted with large, intricate systems [25]; and deploying them where resources are thin—educational tooling, edge devices—runs into practical obstacles [113] that curb their use in enterprise settings.

Security risks are another major concern. Generative models may introduce vulnerabilities or insecure coding patterns that are difficult to detect at the development stage [46], so these models need to be more closely combined with security-aware learning and verification mechanisms [69,114]. At wider scale, shared models and dependencies can also create correlated vulnerabilities across otherwise separate software systems, increasing the importance of supply-chain monitoring and staged human approval.

9.7. Future Research Directions

Generative artificial intelligence research for software engineering is expected to focus on several important directions. One of the key directions is the autonomous software engineering system [20]. Such systems are designed to realise end-to-end automation of software generation, testing and maintenance, and minimise manual intervention [3,99,115].

Another important direction is verifiable code generation. This involves combining formalised methods with generative models to ensure the correctness of the code [35]. This is especially important for security-critical systems that are crucial to reliability [67].

The third direction is DevOps automation based on the Large Language Model (LLM) [21], which includes intelligent automation of deployment, monitoring, testing and maintenance processes [22]. In addition, researchers are also exploring new computational paradigms such as quantum-classical hybrid methods to achieve parameter-efficient code generation.

Human-centred AI engineering, finally, will stay at the centre of the research agenda [2]: the aim is a working balance between automation and human control, in a development environment that people can inspect, interpret and reasonably trust [15,116].

9.8. Review Scope and Limitations

Several limitations should qualify the interpretation of this review. First, the corpus is deliberately journal-focused. This improves consistency in article maturity and reporting, but it excludes peer-reviewed conference literature from major software engineering venues such as ICSE, ESEC/FSE, ASE and ISSTA. The review may therefore underrepresent fast-moving results in areas such as test generation, program repair and agentic development.

Second, although the databases, publication window, search string and inclusion/exclusion rules are reported, the retained review record does not provide a PRISMA-style stage-by-stage count, the exact search execution date or an inter-rater agreement statistic. The review is therefore best understood as a structured literature synthesis rather than a fully protocol-reproducible systematic review. Future updates can strengthen reproducibility through a preregistered protocol, dual screening and explicit study-quality appraisal [98,102].

Third, the lifecycle framework and four-dimension taxonomy are conceptual organising devices based on established software engineering stages and recurring themes in the reviewed literature. They have not been externally validated, and alternative classifications are possible. Finally, the field is moving rapidly; as Figure 8 shows, most studies in the corpus were published in 2025–2026. These limitations do not invalidate the synthesis, but they bound the strength and generalisability of its claims.

10. Conclusions

Drawing on a structured review of 113 peer-reviewed journal studies, this review has taken stock of where generative AI now stands in software engineering. The picture that emerges is of a technology that has outgrown its role as a helper on scattered tasks: what it now reshapes is the life cycle itself, from requirements analysis through to maintenance.

The development paradigm, the analysis suggests, is being remade at its foundations. Traditional deterministic workflows are gradually giving way to probabilistic and adaptive processes; in these new processes, software artifacts are continuously generated, evaluated, and refined through human-AI interaction. This change not only affects technical practice, but also changes the cognitive structure of development work—the role of developers is increasingly changing from direct code writers to system arrangers.

At the same time, the study also points out that there is still a gap between the current technical capacity and the demand for production-level applications. Problems such as code "hallucination", security vulnerabilities and lack of formal correctness guarantee have not been solved. The current evaluation practice still relies mainly on indicators that measure structural characteristics, and cannot comprehensively evaluate the correctness of the semantic level. In addition, there is still a lack of a unified framework for the quality assurance of AI-generated code, and it is impossible to integrate correctness, reliability, security, maintainability and efficiency into a coherent evaluation model.

The research results also show that human-computer collaboration brings new risks while improving productivity. Trust calibration, transparency, and the possible weakening of basic programming skills require continuous attention from the research and education circles. In addition, ethical issues involving intellectual property rights, data privacy and environmental sustainability have also made the responsible deployment of generative artificial intelligence in the field of software engineering more complicated.

Several directions ahead look genuinely promising. End-to-end autonomous software engineering, with human intervention pared to a minimum, remains the long-horizon goal; verifiable code generation—formal methods married to neural models—will matter most where safety is critical; and human-centred AI engineering, holding automation and effective oversight in balance, is what will make any of this sustainable. A unified evaluation framework and a workable governance structure are needed alongside, so that the field's speed does not outrun the standards of quality, safety and ethics that modern software engineering practice demands.

Funding

This research received no external funding.

Data Availability Statement

The data supporting the findings of this study are available from the corresponding author upon reasonable request.

Acknowledgments

The authors would like to thank all individuals who provided valuable assistance and support during the preparation of this manuscript.

Conflicts of Interest

The author declares no conflict of interest.

Use of AI and AI-Assisted Technologies

During the preparation of this manuscript, the authors used ChatGPT (OpenAI) to improve the clarity, grammar, and overall readability of the English language. After using this tool, the authors carefully reviewed and edited the content as needed and take full responsibility for the content of the published article.

References

1. Gao, C.; Hu, X.; Gao, S.; et al. The Current Challenges of Software Engineering in the Era of Large Language Models. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–30. https://doi.org/10.1145/3712005.
2. Zhang, S.; Xing, Z.; Guo, R.; et al. Empowering Agile-Based Generative Software Development through Human-AI Teamwork. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–46. https://doi.org/10.1145/3702987.
3. Ahmed, M.A.; Anwar, M.W.; Butt, W.H.; et al. VeriGen: An LLM-Augmented Framework for End-to-End Automation of Software Development Lifecycle–From Requirements Specifications to Code Generation. *IEEE Access* **2026**, *14*, 63962–63990. https://doi.org/10.1109/access.2026.3686308.
4. Terragni, V.; Vella, A.; Roop, P.; et al. The Future of AI-Driven Software Engineering. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–20. https://doi.org/10.1145/3715003.

5. Jiang, J.; Wang, F.; Shen, J.; et al. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 1–72. https://doi.org/10.1145/3747588.
6. Truong, J.; Nguyen, V.; Nguyen, T.T. An overview of evaluation and enhancement methods for code generation by large language models. *Inf. Softw. Technol.* **2026**, *197*, 108185. https://doi.org/10.1016/j.infsof.2026.108185.
7. Faraji, A.; Pombo, N. AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges. *IEEE Access* **2025**, *13*, 183296–183313. https://doi.org/10.1109/access.2025.3623944.
8. Almanasra, S.; Suwais, K. Analysis of ChatGPT-Generated Codes Across Multiple Programming Languages. *IEEE Access* **2025**, *13*, 23580–23596. https://doi.org/10.1109/access.2025.3538050.
9. Tang, Y.; Liu, Z.; Zhou, Z.; et al. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Trans. Softw. Eng.* **2024**, *50*, 1340–1359. https://doi.org/10.1109/tse.2024.3382365.
10. Dakhel, A.M.; Nikanjam, A.; Majdinasab, V.; et al. Effective test generation using pre-trained Large Language Models and mutation testing. *Inf. Softw. Technol.* **2024**, *171*, 107468. https://doi.org/10.1016/j.infsof.2024.107468.
11. Zhou, X.; Liang, P.; Zhang, B.; et al. Exploring the problems, their causes and solutions of AI pair programming: A study on GitHub and Stack Overflow. *J. Syst. Softw.* **2025**, *219*, 112204. https://doi.org/10.1016/j.jss.2024.112204.
12. Sun, X.; Ståhl, D.; Sandahl, K.; et al. Quality assurance of LLM-generated code: Addressing non-functional quality characteristics. *J. Syst. Softw.* **2026**, *238*, 112885. https://doi.org/10.1016/j.jss.2026.112885.
13. Zheng, Z.; Ning, K.; Zhong, Q.; et al. Towards an understanding of large language models in software engineering tasks. *Empir. Softw. Eng.* **2025**, *30*, 50. https://doi.org/10.1007/s10664-024-10602-0.
14. Dong, Y.; Ding, J.; Jiang, X.; et al. CodeScore: Evaluating Code Generation by Learning Code Execution. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–22. https://doi.org/10.1145/3695991.
15. Basha, M.; Rodríguez-Pérez, G. Trust, transparency, and adoption in generative AI for software engineering: Insights from Twitter discourse. *Inf. Softw. Technol.* **2025**, *186*, 107804. https://doi.org/10.1016/j.infsof.2025.107804.
16. Walczak, J.; Tomalak, P.; Laskowski, A. Impact of code context and prompting strategies on automated unit test generation with modern general-purpose large language models. *J. Syst. Softw.* **2026**, *237*, 112834. https://doi.org/10.1016/j.jss.2026.112834.
17. Pendyala, V.S.; Thakur, N.B. Performance and interpretability analysis of code generation large language models. *Neurocomputing* **2025**, *656*, 131461. https://doi.org/10.1016/j.neucom.2025.131461.
18. Tambon, F.; Moradi-Dakhel, A.; Nikanjam, A.; et al. Bugs in large language models generated code: An empirical study. *Empir. Softw. Eng.* **2025**, *30*, 65. https://doi.org/10.1007/s10664-025-10614-4.
19. Liu, F.; Liu, Y.; Shi, L.; et al. Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code. *IEEE Trans. Softw. Eng.* **2026**, *52*, 1037–1055. https://doi.org/10.1109/tse.2026.3657432.
20. Otoum, N.; Elkhalili, N. Methods and Techniques of Agentic Software Engineering: A Systematic Literature Review. *IEEE Access* **2026**, *14*, 7443–7465. https://doi.org/10.1109/access.2026.3652325.
21. Ljaljevic, S.; Jorba, J.; Iserte, S. Exploring the role of Large Language Models in High-Performance Computing programming: A survey. *Future Gener. Comput. Syst.* **2026**, *184*, 108618. https://doi.org/10.1016/j.future.2026.108618.
22. Valdivia, J.; Mauricio, D.; Guzmán-Jiménez, R. Generative Artificial Intelligence for Software Development Using ISO/IEC 29110 Basic Profile: Gaps and Opportunities. *IEEE Access* **2026**, *14*, 31358–31377. https://doi.org/10.1109/access.2026.3659713.
23. Liu, S.; Guo, D.; Zhang, J.; et al. An Empirical Study of Exploring the Capabilities of Large Language Models in Code Learning. *IEEE Trans. Softw. Eng.* **2025**, *51*, 3088–3102. https://doi.org/10.1109/tse.2025.3609876.
24. Yang, G.; Zhou, Y.; Chen, X.; et al. Chain-of-Thought in Neural Code Generation: From and for Lightweight Language Models. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2437–2457. https://doi.org/10.1109/tse.2024.3440503.
25. Liu, J.; Li, S.; Liu, Z.; et al. Towards multi-language repository-level code generation: From-scratch to guided tasks. *Neurocomputing* **2026**, *679*, 133204. https://doi.org/10.1016/j.neucom.2026.133204.
26. Weyssow, M.; Zhou, X.; Kim, K.; et al. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–25. https://doi.org/10.1145/3714461.
27. Mahade Hasan, M.; Waseem, M.; Kemell, K.K.; et al. Assessing small language models for code generation: An empirical study with benchmarks. *J. Syst. Softw.* **2026**, *236*, 112815. https://doi.org/10.1016/j.jss.2026.112815.
28. Nguyen, T.T.; Vu, T.T.; Vo, H.D.; et al. An empirical study on capability of Large Language Models in understanding code semantics. *Inf. Softw. Technol.* **2025**, *185*, 107780. https://doi.org/10.1016/j.infsof.2025.107780.
29. Li, Y.; Shi, J.; Zhang, Z. An Approach for Rapid Source Code Development Based on ChatGPT and Prompt Engineering. *IEEE Access* **2024**, *12*, 53074–53087. https://doi.org/10.1109/access.2024.3385682.
30. Liu, Z.; Tang, Y.; Luo, X.; et al. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE Trans. Softw. Eng.* **2024**, *50*, 1548–1584. https://doi.org/10.1109/tse.2024.3392499.
31. Lin, Z.; Liu, Z.; Zhou, M.; et al. Effective Fine-tuning for Low-resource Languages: A Case Study of Cangjie. *Empir. Softw. Eng.* **2026**, *31*, 150. https://doi.org/10.1007/s10664-026-10878-4.

32. Li, S.; Cheng, Y.; Chen, J.; et al. Performance analysis of AI-generated code: A case study of Copilot, Copilot Chat, CodeLlaMa, and DeepSeek-Coder models. *Empir. Softw. Eng.* **2026**, *31*, 62. https://doi.org/10.1007/s10664-025-10776-1.
33. Shi, Y.; Bangash, A.A.; Fallahzadeh, E.; et al. HAFix: History-augmented large language models for bug fixing. *Empir. Softw. Eng.* **2026**, *31*, 104. https://doi.org/10.1007/s10664-026-10833-3.
34. Kang, S.; Yoon, J.; Askarbekkyzy, N.; et al. Evaluating Diverse Large Language Models for Automatic and General Bug Reproduction. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2677–2694. https://doi.org/10.1109/tse.2024.3450837.
35. Newcomb, L.; Newcomb, A.; Ochoa, O. Preconditions and Postconditions as Design Constraints for LLM Code Generation. *IEEE Access* **2025**, *13*, 184259–184274. https://doi.org/10.1109/access.2025.3625819.
36. Zhang, B.; Zhang, L.; Luo, Z.; et al. Are They All Good? Evaluating the Quality of CoTs in LLM-Based Code Generation. *IEEE Trans. Softw. Eng.* **2026**, *52*, 1928–1943. https://doi.org/10.1109/tse.2026.3676295.
37. Ouyang, S.; Zhang, J.M.; Harman, M.; et al. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–28. https://doi.org/10.1145/3697010.
38. Bui, T.D.; Vu, T.T.; Nguyen, T.T.; et al. Correctness assessment of code generated by Large Language Models using internal representations. *J. Syst. Softw.* **2025**, *230*, 112570. https://doi.org/10.1016/j.jss.2025.112570.
39. Costa, L.A.; Fontão, A.; Santos, R.P.D.; et al. Applying generative artificial intelligence for vulnerability fixing in a proprietary software ecosystem. *J. Syst. Softw.* **2026**, *234*, 112723. https://doi.org/10.1016/j.jss.2025.112723.
40. Qian, X.; AlOmar, E.A. Can large language models identify and refactor code clones? An empirical study. *J. Syst. Softw.* **2026**, *234*, 112717. https://doi.org/10.1016/j.jss.2025.112717.
41. Dong, Y.; Jiang, X.; Jin, Z.; et al. Self-Collaboration Code Generation via ChatGPT. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–38. https://doi.org/10.1145/3672459.
42. Ahmad, A.; Naeem, M. Boosting Test Generation in Industrial Codebases: A Comparative Study of Base and Fine-Tuned LLMs. *IEEE Access* **2026**, *14*, 73436–73454. https://doi.org/10.1109/access.2026.3690571.
43. Zhu, Y.; Liu, C.; He, X.; et al. AdaCoder: An Adaptive Planning and Multi-Agent Framework for Function-Level Code Generation. *IEEE Trans. Softw. Eng.* **2026**, *52*, 631–650. https://doi.org/10.1109/tse.2025.3642621.
44. Ye, S.; Sun, Z.; Wang, G.; et al. Prompt Alchemy: Automatic Prompt Refinement for Enhancing Code Generation. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2472–2493. https://doi.org/10.1109/tse.2025.3589634.
45. Khojah, R.; de Oliveira Neto, F.G.; Mohamad, M.; et al. The Impact of Prompt Programming on Function-Level Code Generation. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2381–2395. https://doi.org/10.1109/tse.2025.3587794.
46. Zhao, J.; Sun, Y.; Huang, C.; et al. Towards Secure Code Generation with LLMs: A Study on Common Weakness Enumeration. *IEEE Trans. Softw. Eng.* **2025**, *51*, 3507–3523. https://doi.org/10.1109/tse.2025.3619281.
47. Palla, D.; Slaby, A. Evaluation of Generative AI Models in Python Code Generation: A Comparative Study. *IEEE Access* **2025**, *13*, 65334–65347. https://doi.org/10.1109/access.2025.3560244.
48. Ságodi, Z.; Siket, I.; Ferenc, R. Methodology for Code Synthesis Evaluation of LLMs Presented by a Case Study of ChatGPT and Copilot. *IEEE Access* **2024**, *12*, 72303–72316. https://doi.org/10.1109/access.2024.3403858.
49. Rizqullah, M.; Albassam, E. Model-Agnostic Empirical Evaluation of Test-Driven Prompt Engineering on Improving Accuracy and Efficiency in Large Language Models Python Code Generation. *IEEE Access* **2026**, *14*, 22801–22821. https://doi.org/10.1109/access.2026.3662817.
50. Esnaola, L.M.; Ramón, H.D.; Lanzarini, L.C. Can generative AI bridge the gap? A quasi-experimental study of non-programmers with AI vs. programmers without AI. *Empir. Softw. Eng.* **2026**, *31*, 75. https://doi.org/10.1007/s10664-026-10813-7.
51. Chan, P.Y.P.; Keung, J.; Yang, Z. Effectiveness of symmetric metamorphic relations on validating the stability of code generation LLM. *J. Syst. Softw.* **2025**, *222*, 112330. https://doi.org/10.1016/j.jss.2024.112330.
52. Yang, G.; Zhou, Y.; Cheng, W.; et al. Less Is More: DocString Compression in Code Generation. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 1–31. https://doi.org/10.1145/3735636.
53. Dong, Y.; Kong, L.; Zhang, L.; et al. A search-and-fill strategy to code generation for complex software requirements. *Inf. Softw. Technol.* **2025**, *177*, 107584. https://doi.org/10.1016/j.infsof.2024.107584.
54. Delgado, D.; Burgueño, L.; Clarisó, R. A framework for assessing the capabilities of code generation of constraint domain-specific languages with large language models. *J. Syst. Softw.* **2026**, *238*, 112871. https://doi.org/10.1016/j.jss.2026.112871.
55. Ko, A.J.; Myers, B.A.; Coblenz, M.; et al. A Retrospective on How Developers Seek, Relate, and Collect Information About Code. *IEEE Trans. Softw. Eng.* **2025**, *51*, 848–851. https://doi.org/10.1109/tse.2025.3534071.
56. Patel, M.S.; Dhawan, R.; Bramhanwade, L.; et al. Code Comprehension Gaps in AI-Assisted Development: A Conceptual Framework for Knowledge Erosion Patterns. *IEEE Access* **2026**, *14*, 66740–66757. https://doi.org/10.1109/access.2026.3689574.
57. Bhandari, G.; Gavric, N.; Shalaginov, A. Generating vulnerability security fixes with Code Language Models. *Inf. Softw. Technol.* **2025**, *185*, 107786. https://doi.org/10.1016/j.infsof.2025.107786.

58. Ahn, J.; Lee, S.; Kang, S. An MCP Tool for Bug Localization by Leveraging Modification Frequencies of Similar Bug Reports. *IEEE Access* **2026**, *14*, 25242–25263. https://doi.org/10.1109/access.2026.3665337.
59. Balim, C.; Karasekreter, N.; Aslan, Ö. Automatic multi-language analysis of SOLID compliance via machine learning algorithms. *Inf. Softw. Technol.* **2026**, *192*, 108013. https://doi.org/10.1016/j.infsof.2026.108013.
60. Yu, X.; Liu, L.; Hu, X.; et al. Fight Fire with Fire: How Much Can We Trust ChatGPT on Source Code-Related Tasks? *IEEE Trans. Softw. Eng.* **2024**, *50*, 3435–3453. https://doi.org/10.1109/tse.2024.3492204.
61. Zhuo, T.Y.; He, J.; Sun, J.; et al. Identifying and Mitigating API Misuse in Large Language Models. *IEEE Trans. Softw. Eng.* **2026**, *52*, 855–873. https://doi.org/10.1109/tse.2026.3651566.
62. Trong Vu, T.; Bui, T.D.; Nguyen, T.T.; et al. Model-agnostic quality assessment for LLM-generated code via dynamic internal representation selection. *J. Syst. Softw.* **2026**, *237*, 112852. https://doi.org/10.1016/j.jss.2026.112852.
63. Li, X.; Yuan, S.; Gu, X.; et al. Few-shot code translation via task-adapted prompt learning. *J. Syst. Softw.* **2024**, *212*, 112002. https://doi.org/10.1016/j.jss.2024.112002.
64. De Siano, G.D.; Fasolino, A.R.; Sperlí, G.; et al. Translating code with Large Language Models and human-in-the-loop feedback. *Inf. Softw. Technol.* **2025**, *186*, 107785. https://doi.org/10.1016/j.infsof.2025.107785.
65. Zhao, Z.; Fard, F. Do Current Language Models Support Code Intelligence for R Programming Language? *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–39. https://doi.org/10.1145/3735635.
66. Eagal, A.; Stolee, K.; Ore, J. Analyzing the dependability of Large Language Models for code clone generation. *J. Syst. Softw.* **2025**, *230*, 112548. https://doi.org/10.1016/j.jss.2025.112548.
67. Umer, M.M. Comparative Analysis of the Code Generated by Popular Large Language Models (LLMs) for MISRA C++ Compliance. *IEEE Access* **2025**, *13*, 194815–194831. https://doi.org/10.1109/access.2025.3633086.
68. Tony, C.; Díaz Ferreyra, N.E.; Mutas, M.; et al. Prompting Techniques for Secure Code Generation: A Systematic Investigation. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–53. https://doi.org/10.1145/3722108.
69. Huang, L.; Yan, M.; Yin, T.; et al. Steer Your Model: Secure Code Generation with Contrastive Decoding. *IEEE Trans. Softw. Eng.* **2026**, *52*, 809–834. https://doi.org/10.1109/tse.2025.3650127.
70. Li, D.; Shu, S.; Yan, M.; et al. Improving Co-Decoding Based Security Hardening of Code LLMs Leveraging Knowledge Distillation. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2634–2650. https://doi.org/10.1109/tse.2025.3591791.
71. Wang, Q.; Wang, J.; Li, M.; et al. A Roadmap for Software Testing in Open-Collaborative and AI-Powered Era. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–17. https://doi.org/10.1145/3709355.
72. Kumar, C.; Sri Ponaka, U.; Naidu, P.V.L.N.; et al. AI-Powered Unit Test Generation via Multi-LLM Chaining: A Case Study With GPT-4o, Gemini, and Claude-3.5. *IEEE Access* **2025**, *13*, 204058–204071. https://doi.org/10.1109/access.2025.3637221.
73. Yang, Z.; Wang, L. IntelliUnitGen: A Unit Test Case Generation Framework Based on the Integration of Static Analysis and Prompt Learning. *IEEE Access* **2025**, *13*, 177760–177784. https://doi.org/10.1109/access.2025.3615990.
74. Alagarsamy, S.; Tantithamthavorn, C.; Takerngsaksiri, W.; et al. Enhancing large language models for text-to-testcase generation. *J. Syst. Softw.* **2025**, *230*, 112531. https://doi.org/10.1016/j.jss.2025.112531.
75. Trong, M.P.; Tung Son, N.; Hai, L.N.; et al. A Business Process-Centric Approach for LLM-Driven Functional Test Generation. *IEEE Access* **2026**, *14*, 44120–44146. https://doi.org/10.1109/access.2026.3671629.
76. Sisomboon, W.; Kaewyotha, J.; Songpan, W. Automated Software Test Case Generation Using Directional Partially Weighted Ensemble Large Language Models with Retrieval-Augmented Generation (RAG). *IEEE Access* **2026**, *14*, 31124–31145. https://doi.org/10.1109/access.2026.3667925.
77. Karpurapu, S.; Myneni, S.; Nettur, U.; et al. Comprehensive Evaluation and Insights into the Use of Large Language Models in the Automation of Behavior-Driven Development Acceptance Test Formulation. *IEEE Access* **2024**, *12*, 58715–58721. https://doi.org/10.1109/access.2024.3391815.
78. Liu, Z.; Wang, H.; Xu, T.; et al. RAG-Driven multiple assertions generation with large language models. *Empir. Softw. Eng.* **2025**, *30*, 105. https://doi.org/10.1007/s10664-025-10641-1.
79. Haldar, S.; Pierce, M.; Fernando Capretz, L. Exploring the Integration of Generative AI Tools in Software Testing Education: A Case Study on ChatGPT and Copilot for Preparatory Testing Artifacts in Postgraduate Learning. *IEEE Access* **2025**, *13*, 46070–46090. https://doi.org/10.1109/access.2025.3545882.
80. Mehmood, A.; Ilyas, Q.M.; Ahmad, M.; et al. Test Suite Optimization Using Machine Learning Techniques: A Comprehensive Study. *IEEE Access* **2024**, *12*, 168645–168671. https://doi.org/10.1109/access.2024.3490453.
81. Fraser, G.; Arcuri, A. A Retrospective on Whole Test Suite Generation: On the Role of SBST in the Age of LLMs. *IEEE Trans. Softw. Eng.* **2025**, *51*, 874–878. https://doi.org/10.1109/tse.2025.3539458.
82. Ouédraogo, W.C.; Plein, L.; Kaboré, K.; et al. Enriching automatic test case generation by extracting relevant test inputs from bug reports. *Empir. Softw. Eng.* **2025**, *30*, 85. https://doi.org/10.1007/s10664-025-10635-z.
83. Luo, Z.; Du, Q.; Wang, Y.; et al. Enhancing Protocol Fuzzing via Diverse Seed Corpus Generation. *IEEE Trans. Softw. Eng.* **2025**, *51*, 2693–2709. https://doi.org/10.1109/tse.2025.3595396.

84. Nettur, S.B.; Karpurapu, S.; Nettur, U.; et al. Cypress Copilot: Development of an AI Assistant for Boosting Productivity and Transforming Web Application Testing. *IEEE Access* **2025**, *13*, 3215–3229. https://doi.org/10.1109/access.2024.3521407.
85. Fan, X.; Hu, J.; Zheng, W.; et al. SLMFL: Structural and semantic fusion for fault localization based on Small Language Models. *Inf. Softw. Technol.* **2026**, *196*, 108143. https://doi.org/10.1016/j.infsof.2026.108143.
86. Yin, X.; Ni, C.; Li, X.; et al. Improving the ability of pre-trained language model by imparting large language model's experience. *J. Syst. Softw.* **2026**, *234*, 112744. https://doi.org/10.1016/j.jss.2025.112744.
87. Wang, D.S.; Kim, T.I.; Lee, C.G.; et al. Automated Logging with Large Language Models: Eliminating Task-Specific Pretraining. *IEEE Access* **2025**, *13*, 213260–213272. https://doi.org/10.1109/access.2025.3645237.
88. Zhang, J.; Liao, J.; Tang, J.; et al. The Power of Small LLMs: A Multi-Agent for Code Generation via Dynamic Precaution Tuning. *IEEE Trans. Softw. Eng.* **2026**, *52*, 86–99. https://doi.org/10.1109/tse.2025.3632508.
89. Ságodi, Z.; Kolláth, I.; Hegedűs, P.; et al. A Program Synthesis Dataset for LLM Temperature Analysis. *IEEE Access* **2025**, *13*, 184022–184029. https://doi.org/10.1109/access.2025.3625443.
90. Altin, M.; Mutlu, B.; Kilinc, D.; et al. Automated Testing for Service-Oriented Architecture: Leveraging Large Language Models for Enhanced Service Composition. *IEEE Access* **2025**, *13*, 89627–89640. https://doi.org/10.1109/access.2025.3571994.
91. Lin, Z.; Ma, W.; Lin, T.; et al. Open Source AI-based SE Tools: Opportunities and Challenges of Collaborative Software Learning. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–24. https://doi.org/10.1145/3708529.
92. Russo, D. Navigating the Complexity of Generative AI Adoption in Software Engineering. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–50. https://doi.org/10.1145/3652154.
93. Mailach, A.; Gorgosch, D.; Siegmund, N.; et al. "Ok Pal, we have to code that now": Interaction patterns of programming beginners with a conversational chatbot. *Empir. Softw. Eng.* **2025**, *30*, 34. https://doi.org/10.1007/s10664-024-10561-6.
94. Ogenrwot, D.; Businge, J. PatchTrack: A comprehensive analysis of ChatGPT's influence on pull request outcomes. *Empir. Softw. Eng.* **2026**, *31*, 136. https://doi.org/10.1007/s10664-026-10869-5.
95. Ko, E.; Kang, P. Evaluating Coding Proficiency of Large Language Models: An Investigation Through Machine Learning Problems. *IEEE Access* **2025**, *13*, 52925–52938. https://doi.org/10.1109/access.2025.3553870.
96. Nashaat, M.; Miller, J. Towards Efficient Fine-Tuning of Language Models with Organizational Data for Automated Software Review. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2240–2253. https://doi.org/10.1109/tse.2024.3428324.
97. Gəruslu, V.; Jafarov, Z.; Mövsümova, A.; et al. Encouraging responsible GenAI use in software engineering education: A design-oriented model. *J. Syst. Softw.* **2026**, *237*, 112846. https://doi.org/10.1016/j.jss.2026.112846.
98. Li, L.; Duan, L. Human centric innovation at the heart of industry 5.0—exploring research challenges and opportunities. *Int. J. Prod. Res.* **2026**, *64*, 4901–4933. https://doi.org/10.1080/00207543.2025.2462657.
99. Liu, Z.; Zeng, R.; Wang, D.; et al. Agents4PLC: Automating Closed-Loop PLC Code Generation and Verification in Industrial Control Systems Using LLM-Based Agents. *IEEE Trans. Softw. Eng.* **2026**, *52*, 1672–1687. https://doi.org/10.1109/tse.2026.3667895.
100. Fahad, I.A.; Fuad, M.M.N. Can we trust the source? A systematic review of watermarking and attribution for AI-generated code. *Inf. Softw. Technol.* **2026**, *196*, 108157. https://doi.org/10.1016/j.infsof.2026.108157.
101. Martins, M.; Jardim, B.; De Castro Neto, M.; et al. Talking to Data: A Systematic Review of the Rise of Conversational Agents for Visual Analytics. *IEEE Access* **2025**, *13*, 208902–208931. https://doi.org/10.1109/access.2025.3638950.
102. Li, J.; Fu, L.; Wang, Y. Generative AI and management analytics: *s*urvey. *J. Manag. Anal.* **2026**, *13*, 217–263. https://doi.org/10.1080/23270012.2026.2652312.
103. Weyssow, M.; Kamanda, A.; Zhou, X.; et al. CodeUltraFeedback: An LLM-as-a-Judge Dataset for Aligning Large Language Models to Coding Preferences. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 1–36. https://doi.org/10.1145/3736407.
104. Jonnala, R.; Yang, J.; Lee, Y.; et al. Measuring and Improving the Efficiency of Python Code Generated by LLMs Using CoT Prompting and Fine-Tuning. *IEEE Access* **2025**, *13*, 119657–119681. https://doi.org/10.1109/access.2025.3585742.
105. Antal, G.; Értekes, L.; Szolnoki, N.; et al. Evaluating Retrieval-Augmented Generation for LLM-Based Vulnerability Detection: An Empirical Study on Real-World Java Vulnerabilities. *IEEE Access* **2026**, *14*, 50878–50891. https://doi.org/10.1109/access.2026.3676577.
106. Tunzina, T.; Mashira, M.; Islam, M.M. GreenAI: A Comparative Analysis of Environmental Efficiency in LLM-Generated Code. *IEEE Access* **2026**, *14*, 18654–18672. https://doi.org/10.1109/access.2026.3658813.
107. Liang, Q.; Zhang, Z.; Liu, C.; et al. Condor: A Code Discriminator Integrating General Semantics with Code Details. *IEEE Trans. Softw. Eng.* **2025**, *51*, 3524–3539. https://doi.org/10.1109/tse.2025.3620145.
108. Liang, Q.; Sun, Z.; Zhao, Y.; et al. Bipartite-Grammar–Aware Pretraining for XML-SQL Code Updating. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 1–26. https://doi.org/10.1145/3731752.
109. Liu, Y.; Foundjem, A.; Khomh, F.; et al. Adversarial attack classification and robustness testing for large language models for code. *Empir. Softw. Eng.* **2025**, *30*, 154. https://doi.org/10.1007/s10664-025-10693-3.

110. Wang, Z. CurricuForge: A Unified Framework for Progressive Code Generation Through Curriculum-Guided Multi-Agent Collaboration with Symbolic Verification. *IEEE Access* **2026**, *14*, 32557–32573. https://doi.org/10.1109/access.2026.3668847.
111. Liao, Y.; Zhang, J.; Keung, J.; et al. Advancing autonomous driving system testing: Demands, challenges, and future directions. *Inf. Softw. Technol.* **2025**, *187*, 107859. https://doi.org/10.1016/j.infsof.2025.107859.
112. Wu, J.; Jiang, Y.; Wei, L.; et al. Question Selection for Multimodal Code Search Synthesis Using Probabilistic Version Spaces. *IEEE Trans. Softw. Eng.* **2025**, *51*, 1724–1744. https://doi.org/10.1109/tse.2025.3565387.
113. Chen, W.C.; Chen, S.Y. TIPS: A Prompt Engineering Framework for Code Classification and Generation on Resource-Constrained Systems. *IEEE Access* **2026**, *14*, 15723–15735. https://doi.org/10.1109/access.2026.3658502.
114. Lim, S.C.; Kim, J.C. Hybrid Quantum-Classical Transfer Learning for Parameter-Efficient Code Generation. *IEEE Access* **2026**, *14*, 65357–65373. https://doi.org/10.1109/access.2026.3687852.
115. Sun, Y.; Li, L.; Yu, Z.; et al. Exploring AI models and applications within a system framework. *Syst. Res. Behav. Sci.* **2025**, *42*, 1163–1180. https://doi.org/10.1002/sres.3036.
116. Kang, Y.; Pu, B.; Kou, Y.; et al. A deep graph network with multiple similarity for user clustering in human–computer interaction. *ACM Trans. Multimed. Comput. Commun. Appl.* **2024**, *20*, 1–20. https://doi.org/10.1145/3549954.