

GENERATIVE ARTIFICIAL INTELLIGENCE AND THE FUTURE OF SOFTWARE ENGINEERING: TRANSFORMING SOFTWARE DEVELOPMENT, TESTING, AND INTELLIGENT AUTOMATION

Irfan Hanif* ¹, Asad Javed², Intikhab Ulfat ³, Manahil Khan ⁴,
Muniba Saleem⁵

¹Research Scholar, School of Architecture, Computing and Engineering, University of East London, England.

Lecturer, Information Engineering Technology, Foundation University School of Science and Technology.

Department of Physics, University of Karachi, Karachi-75270, Pakistan.

irfanhanif@gmail.com , asadjaved3052@gmail.com , iulfat@uok.edu.pk

DOI: <https://doi.org/>

Keywords

Automation, Code generation, Generative artificial intelligence, Intelligent testing, Software development, Software engineering

Article History

Received: 29 March, 2026

Accepted: 22 May, 2026

Published: 23 May, 2026

Copyright @Author

Corresponding Author: *
Irfan Hanif*

Abstract

This study examined the role of generative artificial intelligence in transforming software engineering, particularly software development, testing, and intelligent automation. A descriptive mixed-method research design was used, and data were collected from a sample of 120 respondents, including software developers, software testers, software engineering students, system analysts, project managers, and DevOps or automation engineers. The results showed strong awareness of generative AI among respondents. Generative AI as a tool for faster software development recorded a mean score of 4.28, while its role in changing the future role of software engineers also recorded 4.28. Code completion was the most adopted AI activity, with a mean score of 3.27, followed by code generation with 3.22, documentation writing with 3.07, debugging support with 3.03, test-case generation with 2.93, and DevOps automation with 2.78. The strongest perceived benefit was increased developer productivity, with a mean score of 4.32, followed by reduced repetitive coding tasks with 4.25 and faster software delivery with 4.18. The major challenge was inaccurate AI-generated code, with a mean score of 4.28, followed by security vulnerabilities with 4.23 and overdependence on AI tools with 4.17. These findings suggested that adoption was strongest in routine coding tasks and weaker in advanced automation areas. The study concluded that generative AI supported faster, smarter, and more automated software engineering, but human supervision, training, policy guidance, and security validation remained essential for reliable professional practice.

INTRODUCTION

With the introduction of generative artificial intelligence (AI), we are in the midst of a revolution in the software engineering sector, fundamentally changing the methods through which software systems are designed, developed, tested, maintained, and improved. In the traditional software engineering, requirement analysis, coding, debugging, documentation and QA and maintenance were largely manually done.

This process could be time consuming and resource intensive, particularly with the growth of software systems becoming increasingly large, complex and dynamic. The creation of code, description of technical problems, documentation, design improvement and automated decision-making assistance throughout the software development process, blown up by generative AI, was a new level of intelligent assistance. In recent studies, large language models were used for a variety of software engineering tasks such as code generation, program repair, code testing, code summarization, defect detection, and software development assistance (Hou et al., 2024).

But another dynamic is the developer-software tool relationship, and the introduction of generative AI has been a gamechanger there. The existing software engineering tools were mostly focused on editing, compiling, versioning, testing, and deployment, while the tools with generative AI capabilities served as interactive assistant systems engaging with natural language inputs to produce software outputs. The tools were employed by the developers to convert ideas into code, identify bugs, document code bases they were unfamiliar with, and to remove repetitive coding.

This is a shift in the methods of manual work to human-AI collaboration and the output produced by the Artificial Intelligence was evaluated by the developers. “AI-driven” was thus used to characterize the future of Software Engineering, in which software professionals are increasingly exposed to intelligent systems while interacting with them during the requirements engineering, design, implementation, test and maintenance process (Terragni et al., 2025). Another transformative area of software company operations is productivity and

innovation, which was also revolutionized by Generative AI. Software vendors were driven to deliver reliable software applications fast, at lower development cost, great user experience, and to fill market demands. They were able to accomplish these goals through the use of Generative AI, which ultimately helped to automate repetitive tasks and assisted developers in handling more complex issues. The study on generative AI adoption in software engineering revealed that the tools' functionality was not the sole factor; they also needed to fit into the current software development workflow, had to be ready for the organization, and had to be accepted by the humans (Russo, 2024).

The aim of this study was to delve into the phenomenon of Generative AI and its promise for the future of software engineering. It highlighted the influence of Generative AI in software development, software testing, and intelligent automation. The study also considered opportunities and risks since the software produced by AI must be validated, tested and verified by humans for quality control, security, ethical issues and verification. Generative AI helped speed up

development time, but it also brought some challenges including poor accuracy, eroded security, dependence, and lack of responsibility (Wong et al., 2023).

Background of the Study

The background of this research was related to the large language model training with natural language and programming language. These models were trained from massive software repositories, documentation, programming examples and technical conversations. This enabled them to produce source code, finish incomplete programs, translate source code from one language to another, explain algorithms, and help in the debugging process. With the advent of the recent code generation studies, the potential of large language models for improving the automation of programming tasks has been established, as well as the possibilities for new programs to be created using LLM, at least when evaluated in a number of programming languages and benchmark tasks (Jiang et al., 2026).

One major area of interest was the potential for generative AI, particularly in code generation, where developers could provide a task in natural language and get ideas for code that they can run. This

changed the activity from syntax-driven to human intention and machine-driven (Jiang et al., 2024).

In addition to software testing, the other background topic that was significant to this study was software development. Testing was a very expensive and time-consuming process because it entails designing of test cases, running the tests, detecting the defects and results interpretation. With the advent of Generative AI, there are new opportunities for automated test generation, assertion creation, test repair, and test coverage improvement. The user responses, test cases, and AI-generated code outputs resulted in more accurate programs and more reliable development processes, as shown by the interactive test-driven code generation, according to Fakhoury et al. (2024).

Automated unit test generation was also associated with the usage of generative AI in software testing. The unit tests were used to test a single software component and see whether it works as it should; this was time-consuming and technically difficult to make by hand. The AI generated tests were evaluated for their capacity to automatically generate unit tests,

revealing that in several cases, AI-generated tests provide greater levels of coverage. In the same study, it was concluded that there is a need for a few more tests to be generated as incorrect tests, weak assertions, and irrelevant test cases can reduce software quality (Schäfer et al., 2024).

Research Problem

The primary research problem was that generative artificial intelligence was quickly making its way into the software engineering field, and its future effects on software development, testing and intelligent automation were not known. While numerous organizations have embraced AI coding assistants for their productivity gains, they have remained concerned about reliability, security, maintainability, accountability, and the impact on developer skills. While AI-generated code might save time, it also has the potential to contain unexpected bugs or vulnerabilities if developers blindly adopt the suggested code without verification. The study of software testing using large language models revealed that while AI tools assisted numerous testing activities, there were still issues regarding test quality, oracle generation, correctness, and evaluation standards.

Objectives of the Study

Research Objectives

1. To examine the transformation of software development practices through the adoption of generative artificial intelligence technologies.
2. To investigate the role of generative AI in supporting software engineering activities, including code generation, debugging, documentation, and software maintenance.
3. To evaluate the effectiveness of generative AI in improving software testing processes and enhancing automated test-case generation.
4. To analyze the contribution of generative AI to intelligent automation and DevOps workflows in modern software development environments.

Research Questions

1. How did generative artificial intelligence transform software development practices?
2. How did generative AI support code generation, debugging, documentation, and software maintenance?
3. How did generative AI improve software testing and automated test-case generation?
4. What role did generative AI play in intelligent automation and DevOps-related workflows?

Significance of the Study

The study's impact was significant because it provided insights into the future trajectory of software engineering as a field due to generative AI. Traditional supply of software had been supplemented now by intelligent software systems, generating, testing, explaining and optimizing software artefacts. This study assisted researchers, students, developers, and organizations in a deeper comprehension of the opportunities generative AI provides to unlock greater productivity, minimize repetitive work, drive faster development and innovation. It also helped to spread unprecedented academic knowledge by linking generative AI to software development, software testing, and intelligent automation. It was also a game-changer for software firms and software development teams as they were once again reminded of the need to adopt responsibility. AI capabilities that generated content proved beneficial, but also needed governance, validation, security testing, and human judgment. The study indicated that applying AI tools for programming opened the debate about issues of code quality, trust, security and programmers' responsibility.

Literature Review

Generative AI in Software Development and Code Generation

G-Ai revolutionized software development by enabling the creation, completion, understanding, commenting and helping software developers. Many recent studies indicated that large language models took software engineering away from the rule-based programming environment and into intelligent, prompt-based environments. On the other hand, code generation research highlights the shift in how programmers would use LLM to generate code from natural language textual instruction descriptions and its ability to decrease manual effort in time-consuming programming tasks (Gülmez et al., 2026). An investigation of both pre- and post-conditions indicated that programs generated by LLM models were more reliable when there was explicit control over the generation process through constraints, as constraints assisted the model in generating code that adhered to desired functional properties (Newcomb et al., 2025).

Recent research revealed that the quality of code generated was affected by the fact of the model architecture, the set

of training data, the prompt structure, the complexity of the task, and the support of the programming language. Previous studies on quantized code models demonstrated that smaller and compressed LLMs lowered the barrier for developers who are strapped for compute, while still maintaining similar performance in low-resource programming-language environments (Nyamsuren, 2025). Another performance and interpretability study into code generation demonstrated that not only accuracy measures, but also insight into how the model had reasoned about a solution was important when evaluating the model, and understanding how accurate and trustworthy the solution was in real waterfowl development (Pendyala et al., 2025).

The literature also indicated the emergence of generative AI usage in engineering applications like hardware design and codebase understanding, beyond general software code generation. Research on Generative AI-generated hardware code found improvements in the performance of LLM models for Verilog tasks and recommended using generative AI to support DHD designers as well as SWEs (Pinckney et al., 2025). In addition,

there is research that indicated that for certain programming tasks, employing fine-tuning methods resulted in the enhancement of code generation by code-generation models. Code generation models could be fine-tuned to improve their performance without having to be completely retrained to accomplish the task (Weyssow et al., 2025).

Use GenAI in software testing, debugging and program repair

One of the most significant transformations in software testing prompted by generative AI was that testing became a true interdisciplinary field. One of the most important aspects of the software engineering discipline that was impacted by generative AI was the software testing space itself becoming interdisciplinary. Generating tests was traditionally a time consuming exercise, as testers had to specify the test cases, predict output, run the tests and analyze results. The systematic review evidence in recent studies revealed that LLM was found to facilitate test generation enriching the requirements with tests, generate unit tests, enhance test coverage, and reduce manual testing time (Tasarsu et al., 2026). Recent research on PyDex further demonstrated

that LLMs helped in bug repair on programming task assignments by providing probable repairs and enhancing feedback-driven software rectification proficiency, particularly in structured and educational settings for programming (Zhang et al., 2024).

Generative AI also facilitated the debugging task by using language knowledge and reasoning over program behavior. Explainable automated debugging research demonstrated that scientific bugs debugged with LLM-driven scientific debugging produced hypotheses on the causes of the bugs, interacted with the tools, and generated explanations that assisted developers in better judging if a patch is correct or not, Kang et al., 2025). Additionally, automated program repair literature revealed that there was a shift in organisation from template-based to machine learning / deep learning based approaches and that generative AI was used to realize more semantic and context-aware repair strategies (Dikici et al., 2025). The research studies identified indicated that the use of LLM based repair proved useful in both imperative programming and formal specification languages. When it comes to repairing declarative formal

specifications, LLMs were found to be helpful in various scenarios, such as single-agent or dual-agent approaches, feedback methods, and auto-prompting strategies (Alhanahnah et al., 2025). Previous research in software defect detection also revealed that LLM helps software developers in the identification of defects, learning software patterns, suspicious logic detection, and software quality assurance efforts from the software development and maintenance phases (Chen et al., 2026).

Integrated and intelligent automation, security and responsible adoption

Generative AI also aided the intelligent automation processes, such as code review, engineering decision making AI, design recognition, and quality improvement. Evaluation of design pattern recognition indicated that LLMs used deep learning to learn semantic information and syntactic information from source code to identify designs patterns without the need of extensive predefined manual design patterns (Pandey et al., 2025). When it came to automatic resolution of code issues, the research indicated that future development environments might integrate the value of generative reasoning systems (Patcas et al., 2026) with rule-based tools,

considering how an LLM could be utilized with static analysis results to resolve code-quality issues.

One primary worry with the implementation of generative AI in software engineering is security. A study on security-aware LLM found that the conversation coding assistants did not always alert the users to unsafe code, leading to issues when developers took AI generated answers without verifying them. An investigation with LLM security awareness revealed that the conversational coding assistants were not consistently emitting alerts about insecure code, which poses problems when developers accepted answers without verifying them. (Sajadi et al., 2025) In addition, research on security assertions found that LLM can be used to help with hardware security verification as it could provide security-centric assertions, the generated assertions were not always robust enough to be relied upon by experts to guarantee high reliability and correctness for security-critical engineering. (Kande et al., 2024).

Supported of responsible adoption were also addressed in the literature, and highlighted evaluate, human oversight, and organisational governance. Generative AI

helped lead to higher productivity but results might be flawed, including insecure patterns, weak tests, misunderstandings of requirements and logic flaws. The results of code generation survey research reflected the fact that performance of models was improving very quickly, while in software engineering, reliability, evaluation, benchmarks, and contamination issues were still considered to be crucial (Gülmez et al., 2026). Security investigations and testing revealed that generative AI was most effective when a developer leverages it as a tool to work alongside them, rather than as a standalone solution: quality and accountability, as well as security, required human input when using generative AI (Tasarsu et al., 2026).

Research Methodology

Research Design

The study took a descriptive research design approach to explore the future of software engineering with the support of generative artificial intelligence. The design was chosen as it aimed to explain the impact of generative AI on software development, software testing, and intelligent automation. A descriptive design was used to communicate the

current use, benefits, barriers and potential of generative AI in software engineering practices.

Population of the Study

The population of the study comprised software professionals, Software Engineering learners, Software IT workers, Software Testers, Software Developers, Software System Analysts and Software Project Managers. The participants in this sample were chosen because they had an in-direct or direct involvement with software development, software testing, software maintenance, or intelligent automation. They also gained insight from their experience into the impact of generative AI on contemporary software engineering work.

Sample and Sample Size

The number of respondents of the study were 120 which were selected from software related fields. There were 35 software developers, 25 software testers, 20 software engineering students, 15 system analysts, 15 project managers, and 10 DevOps/automation engineers. This sample size was deemed adequate due to the variety of roles within software engineering and the number of responses,

which found the sample size adequate for meaningful analysis.

Sampling Technique

A purposive sampling method was employed in the study. This technique was chosen due to the research being conducted needing participants to have knowledge or experience in software engineering and generative AI. Random sampling was not appropriate since not all employees in an organization were not familiar with AI-assisted software development, automated testing or intelligent automation. The participants selected in accordance with their professional level, technical skills and relation to software engineering activities. Those who utilized or were aware of tooling like AI coding assistants, automated testing platforms, debugging tools, DevOps systems or software documentation tools had a preference. This allowed for the gathering of insights focused and relevant from knowledgeable respondents.

Data Collection Methods

Structured questionnaire was used to gather primary data. The questionnaire was primarily designed to gather information on different generative AI awareness

dimensions, as well as on the use of generative AI tools in software development, for coding, debugging, software testing, documentation, DevOps automation, expected benefits from generative AI, perceived risks, and expectations for the future. Closed questions were used to obtain measurable data and open questions were used for respondents to give an expression of detailed opinions.

Research Instrument

A structured questionnaire was used as main research instrument. The questionnaire was broken up into sections. The first part was to collect demographic information such as role, experience level and technical background. The second part was dedicated to the awareness of generative AI in Software Engineering. The third part was devoted to generative AI apps in software development, testing and intelligent automation. The last part centered on benefits, challenges, risks and future directions.

Data Collection Procedure

After the research objectives and the questionnaire was decided, the data collection process started. All the respondents were contacted and informed

about the focus of this study. Questionnaire was distributed via forms online and directly through professional networks. The questionnaire was administered with clear instructions before the respondents were requested to answer the questions.

Data Analysis Method

Data analysis of the quantitative data was done using descriptive statistics. Frequency, percentage and mean scores were used to summarize the responses. The methodology allowed them to collect data

regarding awareness of generative AI adoption, perceived software engineering adoption usefulness and barriers to software engineering generative AI adoption. The results were displayed in tables and explained in detail.

Results and Analysis

Demographic Profile of Respondents

The demographic profile helped show whether the selected respondents were suitable for providing information about generative AI and software engineering practices.

Table 1.Demographic Profile of Respondents (N = 120)

Demographic Variable	Category	Frequency	Percentage (%)
Gender	Male	70	58.3
	Female	50	41.7
Age Group	20–30 years	45	37.5
	31–40 years	42	35.0
	41–50 years	22	18.3
	Above 50 years	11	9.2
Professional Role	Software Developers	35	29.2
	Software Testers	25	20.8
	Software Engineering Students	20	16.7
	System Analysts	15	12.5
	Project Managers	15	12.5
	DevOps/Automation Engineers	10	8.3

Demographic Variable	Category	Frequency	Percentage (%)
Work Experience	1–3 years	36	30.0
	4–6 years	40	33.3
	7–10 years	28	23.3
	Above 10 years	16	13.4
Total		120	100.0

The results showed from the Table 1 that a total of 120 respondents participated in the study. Out of the total respondents 58.3% were males and 41.7% were females. This showed that the study was conducted by both males and females and a higher number of males participated in the study compared to females. Whereas both groups were contributing to a diversity of opinions on generative AI for software engineering. As indicated in the age distribution, the age group between 20 to 30 years had the largest number with 37.5% of the participants. The 31–40 year age group was next, accounting for 35.0%. The finding revealed that most of the respondents were young or middle career individuals with

knowledge of modern computer software tools, AI coding assistants, automatic testing frameworks, and digital development environments. The distribution of the professional roles showed that there were more software developers (29.2%) and then software testers (20.8%). The key point here is that developers and testers were directly involved in developing, debugging, generating tests, identifying bugs and improving software quality. The work experience outcomes indicated that around one-third of respondents (33.3%) had 4–6 years of experience, indicating that many respondents had experience with hands-on software engineering tasks.

Table 1: Demographic Profile of Respondents (n = 120)



Source: Survey Data

Figure 1: Demographic Profile of Respondents

Awareness of Generative AI in Software Engineering

This table explained the level of awareness among respondents regarding the use of generative AI in software engineering.

Table 2. Awareness of Generative AI in Software Engineering

Statement	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
Generative AI supported faster software development.	60	42	12	4	2	4.28
Generative AI improved code generation and completion.	58	44	12	4	2	4.27
Generative AI supported automated software testing.	52	46	16	4	2	4.18

Statement	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
Generative AI improved debugging and error detection.	50	44	18	6	2	4.12
Generative AI changed the future role of software engineers.	62	38	14	4	2	4.28

Table 2 revealed that the respondents had good awareness of Generative AI in Software Engineering. The items with the highest means were those that indicated that generative AI has enabled accelerated software development (4.28) and is altering the software engineering profession in the future (4.28). This revealed the respondents' perception of generative AI as a significant influencer in the transformation of software development practices and roles. The mean score of the statement that “generative AI enhanced

code generation and completion” was 4.27. This showed a high understanding of how generative AI is perceived to be useful for writing code, completing functions, suggesting syntax and for programming tasks. The mean scores for the statement on automated software testing and debugging/ error detection are 4.18 and 4.12, respectively. These results also revealed that participants were aware of the potential of generative AI for tasks other than

coding.

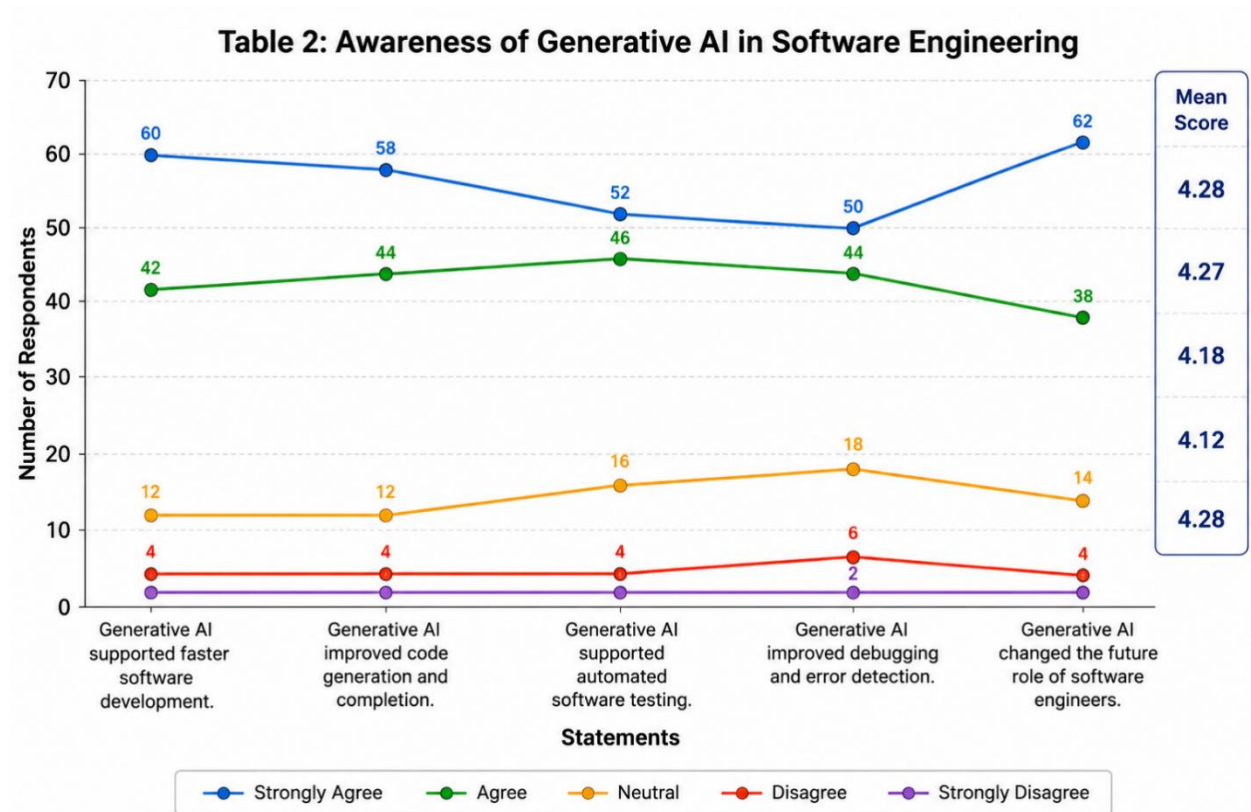


Figure 2: Awareness of Generative AI in Software Engineering

Adoption of Generative AI Tools in Software Engineering

The table explained how frequently generative AI was applied in coding, testing, documentation, debugging, and automation.

Table 3. Adoption of Generative AI Tools in Software Engineering

Software Engineering Activity	Frequently Used	Sometimes Used	Rarely Used	Not Used	Mean Score
Code generation	56	40	18	6	3.22
Code completion	60	38	16	6	3.27
Debugging support	46	42	22	10	3.03
Test-case generation	42	40	26	12	2.93
Documentation writing	50	38	22	10	3.07
DevOps and workflow	34	42	28	16	2.78

Software Engineering Activity	Frequently Used	Sometimes Used	Rarely Used	Not Used	Mean Score
automation					

Table 3 revealed that the most taken code completion use of generative AI had a mean score of 3.27. The mean score for code generation was 3.22 and came in the second place. These results suggested that respondents mostly used generative AI for programming support. This is predicted since the use of AI coding assistants was widespread to complete functionality, suggest code blocks, clarify syntax, and minimize repetitive coding tasks. Documentation writing was given a mean score of 3.07 and debugging support a score of 3.03. These findings revealed a

moderate degree of adoption in the use of the supporting software for supporting software maintenance and software development communication. The test case generation score with a mean of 2.93 and devops and workflow automation with a mean of 2.78 were the lowest scores. These results indicated that adoption was less for more complicated engineering tasks. Precision and ability to integrate with existing systems and rigorous validation were important for test generation and DevOps automation.

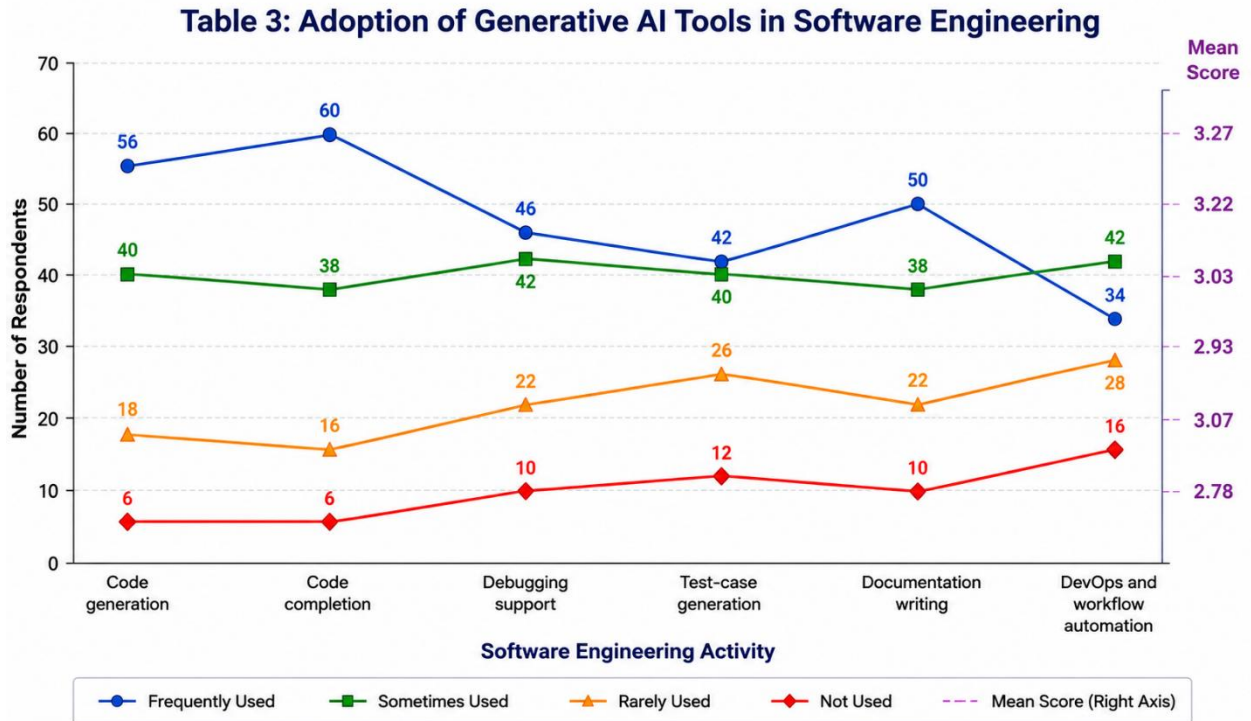


Figure 3: Adoption of Generative AI Tools in Software Engineering

Perceived Benefits of Generative AI in Software Engineering

These benefits included increased productivity, faster development, improved testing, reduced repetitive work, better documentation, and stronger automation.

Table 4. Perceived Benefits of Generative AI in Software Engineering

Benefit	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
Increased developer productivity	64	38	12	4	2	4.32
Reduced repetitive coding tasks	58	42	14	4	2	4.25
Improved speed of software delivery	56	40	16	6	2	4.18
Better support for software testing	48	44	18	8	2	4.07

Benefit	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
Improved documentation quality	52	42	16	8	2	4.12
Supported intelligent automation	46	42	20	10	2	4.00

Table 4 indicated that highest mean score is 4.32 in the case of increased developer productivity. This meant that people were confident that Generative AI contributes positively to the productivity of software professionals. Developers leveraged AI to write code more quickly and better, grasp technical challenges, propose solutions, and save time by avoiding repetitive tasks. Agreement among the respondents was excellent when dealing with reduced repetitive coding tasks, with a mean score of 4.25. This discovery indicated that generative AI could be a valuable tool in repetitive coding tasks like code completion, boilerplates, generating functions, and syntax corrections. The mean scores for improved speed of software delivery is 4.18; improved documentation quality is 4.12. The results in Better support for software testing were 4.07, and intelligent automation 4.00. These results indicated respondents' understanding of the utility of generative AI throughout the software development life cycle.

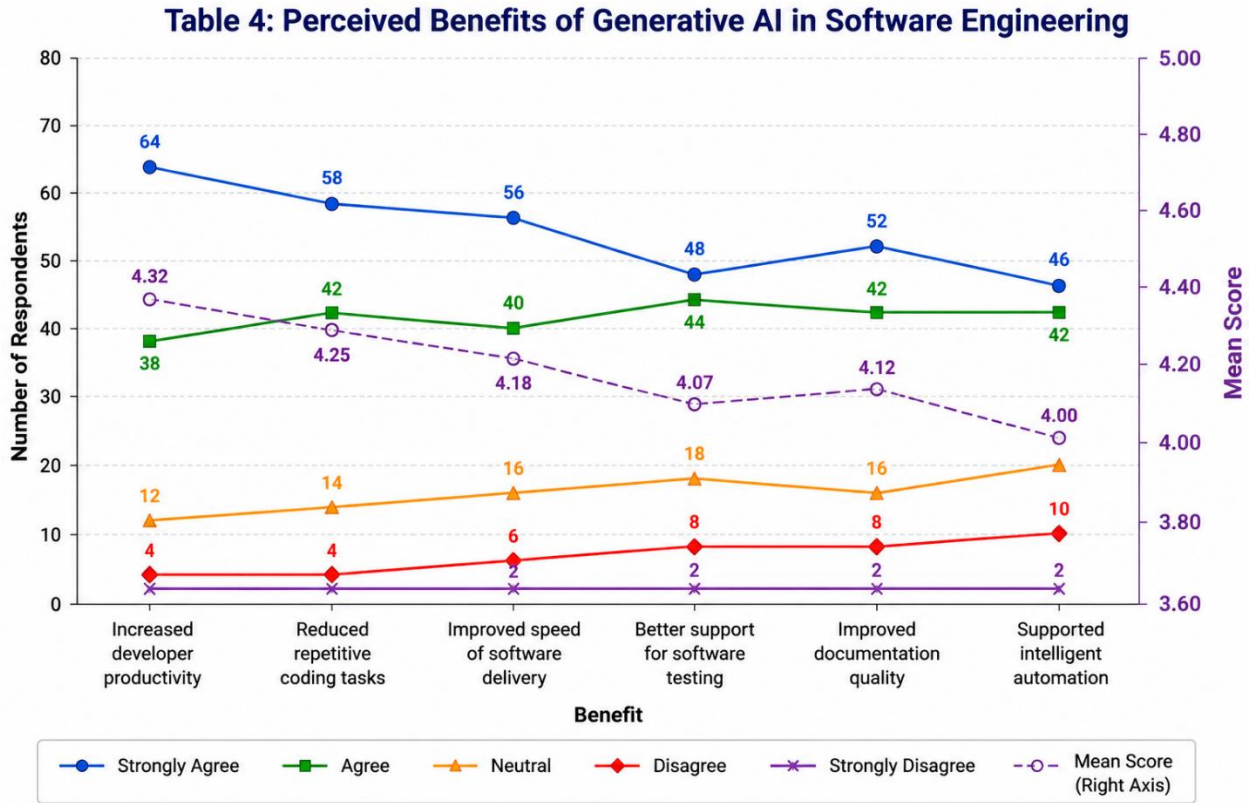


Figure 4: Perceived Benefits of Generative AI in Software Engineering

Challenges and Risks of Generative AI Adoption

Identifying challenges was important because generative AI offered benefits but also introduced risks related to security, reliability, privacy, accuracy, and overdependence.

Table 5. Challenges and Risks of Generative AI Adoption

Challenge/Risk	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
Risk of inaccurate AI-generated code	62	38	14	4	2	4.28
Security vulnerabilities in generated code	58	40	16	4	2	4.23
Overdependence on AI tools	54	42	16	6	2	4.17
Lack of human verification	52	40	20	6	2	4.12
Data privacy and	50	42	18	8	2	4.08

Challenge/Risk	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
confidentiality concerns						
Difficulty integrating AI into workflows	44	42	22	10	2	3.97

Table 5 revealed that the biggest challenge was the risk of wrong code generated by AI: with a mean score of 4.28. The result showed that the respondents were worried about the accuracy of the output produced by AI. While the generated code was helpful for speeding up coding, it was still necessary to review the code manually because it may have contained logical errors, missing functions, faulty assumptions or wrong solutions. The mean score for Security vulnerabilities for generated code was 4.23. Here's some good news, the results demonstrated that

respondents knew they could find unsafe patterns, weak authentication logic, weak input validation, or unsafe dependencies in AI-generated code. Overdependence of AI tools had a mean score of 4.17 and lack of human verification had a mean score of 4.12. The results indicated that the participants felt that AI tools alone should not be relied upon by software engineers. Generative AI was beneficial as an assistant, but it was still important to have human intervention to review, test, secure, and maintain software.

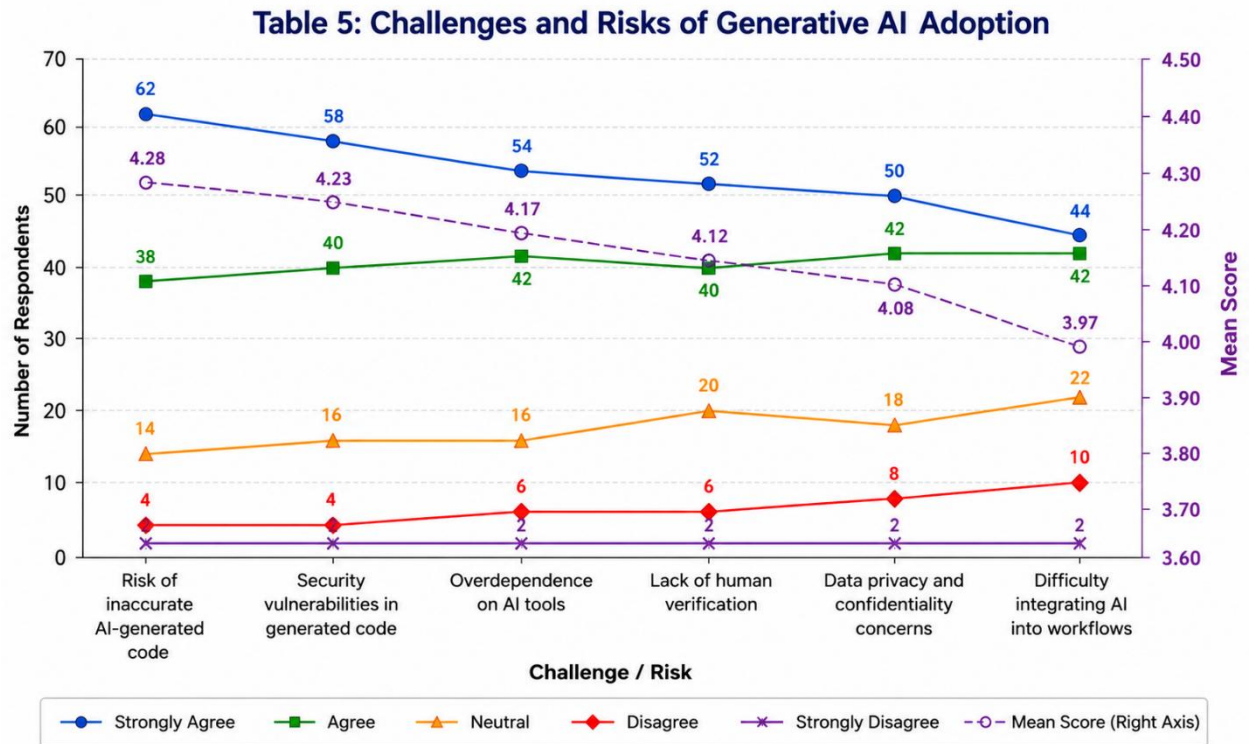


Figure 5: Challenges and Risks of Generative AI Adoption

Future Readiness for Generative AI in Software Engineering

This table explained how prepared respondents and organizations were for the future use of generative AI.

Table 6. Future Readiness for Generative AI in Software Engineering

Statement	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
Software engineers needed training in AI tools.	66	36	12	4	2	4.33
Organizations needed clear AI-use policies.	60	38	14	6	2	4.23
Human supervision remained necessary.	64	34	14	6	2	4.22
AI tools improved future software engineering careers.	54	42	16	6	2	4.17

Statement	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Mean Score
Ethical guidelines were needed for AI-assisted development.	58	38	16	6	2	4.20
AI would become a regular part of software teams.	56	40	16	6	2	4.18

As per the Table 6, the highest mean score has been obtained on the statement “Software engineers require training on AI tools” with a mean score of 4.33. This suggested that the people faced the strong belief that they needed to be skilled at using generative AI responsibly in the future. Training was deemed as essential for quick writing, code review, AI supported testing, debugging, documentation, and security verification. When it comes to policies for the use of AI in organizations, the mean score was 4.23, indicating that the need for human

supervision is still required. The results indicated that the respondents agreed that the use of generative AI requires rules and human control. Because of the potential privacy, security, appropriate use, quality assurance, and ownership issues, AI-use policies were required to train staff about these concerns. Ethical guidelines scored a mean of 4.20 and the statement that AI would be a part of software teams is 4.18. This indicated that respondents believed that generative AI will be a standard software engineering practice in the future.

Table 6: Future Readiness for Generative AI in Software Engineering

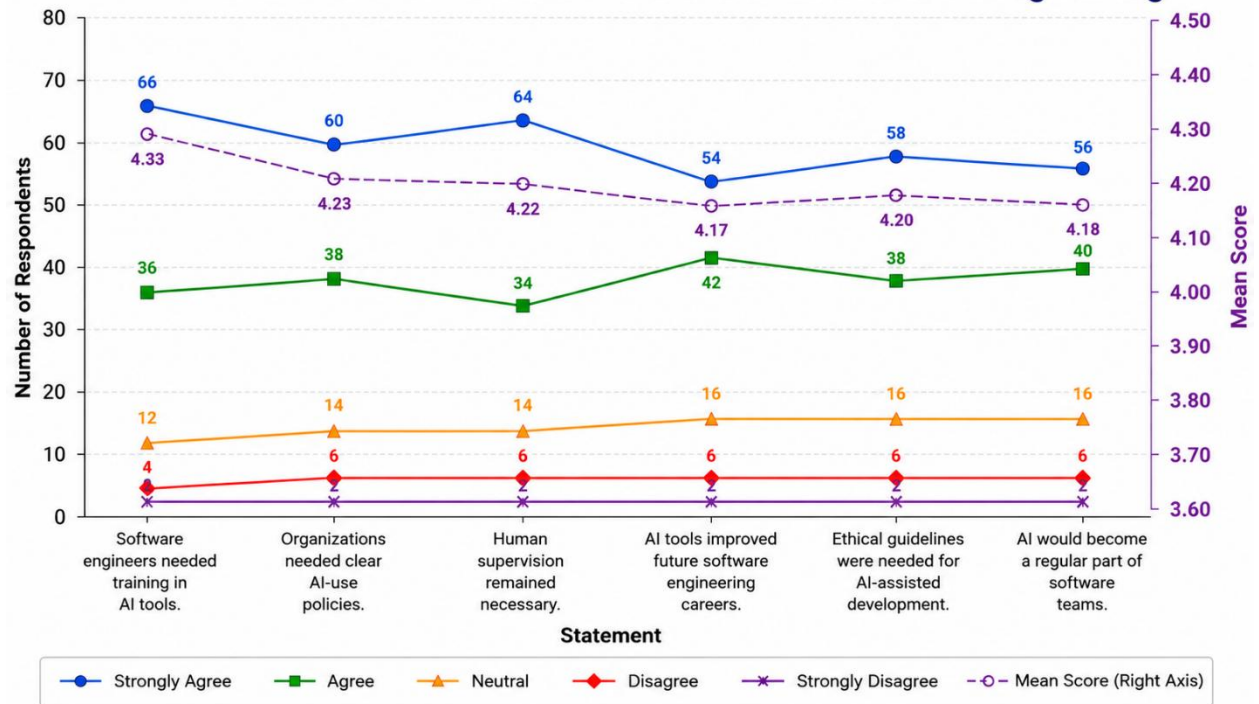


Figure 6: Future Readiness for Generative AI in Software Engineering

Discussion

The results revealed that the respondents had high knowledge about generative AI and its influence on software engineering. They understood how Generative AI would help them develop faster, generate code, debug, test, document and automate intelligently. This means that rather than being a novelty, the use of generative AI was transitioning from a research laboratory to real applications within software processes. This was mirrored by the high awareness scores, which indicated more usage of AI coding assistants in real coding environments, where developers

were guided through coding challenges, found solutions, or completed implementation tasks using the conversational systems (Ehsani et al., 2026; Ziegler et al., 2024).

It was also discovered that code completion and code generation were the primary applications for the use of generative AI. This indicated that respondents believed in the ability of AI tools more in direct programming assistance than in complex engineering decisions. Immediate value could be gained from the code generation and code completion features as they cut down

repetitive syntax writing tasks, saved on boilerplate code, explained logic, and sped up the programmers from idea to implementation. Some previous works also found improvements in productivity when developers used large language models for specific coding tasks with tight control over the output generated (Weber et al., 2024; Cirett-Galán et al., 2024).

The use of generative AI for software testing seemed more moderate compared to coding, indicating that the space of software testing was more sensitive. Participants had prior knowledge of the potential for AI to create test cases, identify edge cases and aid in defect detection, but were also aware that weak tests could lead to false confidence. Additionally, it was found that ChatGPT and Copilot could help with the testing artifacts, but manual review was still required for checking accuracy, coverage, and relevance (Halдар et al., 2025; Ardic et al., 2026).

The number one desired benefit was higher developer productivity. This revealed that the respondents felt generative AI was helpful in saving time, reducing repetitive work and enhancing the development efficiency. Productivity, however, did need to be interpreted

carefully as more output did not necessarily mean higher software quality. Many software quality activities could be facilitated by generative AI solutions, but the industrial readiness was found to rely on the solutions validation, process integration, maturity, and contextual accuracy (Gheveter et al., 2026; Najmi & El-Dosuky, 2025).

The results demonstrated that the generative AI tools were also helpful in requirements engineering, documentation, planning of workflows and early project decisions-making. This was crucial, as the advent of software engineering began with requirements understanding, scope estimation, and turning the needs of stakeholders to technical details. Requirements extraction, Reduction of ambiguities, Identification of terminologies and completeness of requirements were supported by LLMs, and still required human validation (Hemmat et al., 2025; Luitel et al., 2024).

The findings related to risk indicated a number of problems with the AI-generated code that include incorrect code, vulnerabilities, privacy issues, reliance on AI, and inadequate human checks. Even when the code is correct, it

may have faulty assumptions, incomplete exception handling, bad implementation patterns or be otherwise logically incorrect. There are also a number of studies that demonstrate the ability of LLMs to aid the reproduction of bugs and in performing static analysis tasks, but they found that performance is different for each LLM and depending on the programming context (Venkatesh et al., 2025; Kang et al., 2024).

Conclusion

Based on the study, generative AI has been found to have a significant effect on the future of software engineering. The results demonstrated how generative AI helps in software development through various aspects, such as code generation, code completion, debugging, documentation, testing, and intelligent automation. Respondents identified generative AI as a valuable component in boosting productivity, minimizing repetitive coding efforts, expedite software delivery, and assist software quality efforts. The study also revealed that that generative artificial intelligence has also altered the nature of software engineers, transforming them from just a code writer to a code reviewer, supervisor, and evaluator of AI-generated content.

Recommendations

Software companies were advised to be careful and take a systematic approach when implementing generative AI tools. AI-generated code should not be directly used in actual software systems without review, testing and validation by companies. AI-generated content needs to be code-reviewed, debugged, and thoroughly checked for security and quality assurance before implementation. This will minimize the chances of wrong code, undetected errors and software weaknesses.

Future Direction

Future studies should explore the long-term implications of Generative AI on software engineering productivity, software quality, and developer skills. Further research is needed to establish the difference between the speed, accuracy, reliability, cost, and maintainability of AI-assisted development and the standard methods of development. This type of study could shed light on areas where generative AI was most beneficial and where traditional human Intelligence was still more crucial.

References

- Alhanahnah, M., Bagheri, H., Ghassemi, M., & Ahmed, I. (2025). An empirical evaluation of pre-trained large language models for repairing declarative formal specifications. *Empirical Software Engineering*, 30, <https://doi.org/10.1007/s10664-025-10687-1>
- Ardic, B., Le Dilavrec, Q., & Zaidman, A. (2026). How students use generative AI for software testing: An observational study. *Empirical Software Engineering*, 31
- Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M. H., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., & Jangda, A. (2023). MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7), 3675–3691. <https://doi.org/10.1109/TSE.2023.3267446>
- Chen, Y., Wang, X., Li, Z., Liu, H., Zhang, J., & Wang, Q. (2026). Software defect detection using large language models: A literature review. *Frontiers of Computer Science*, 20. <https://doi.org/10.1007/s11704-025-40672-2>
- Cirett-Galán, F., Torres-Peralta, R., Navarro-Hernández, R., Ochoa-Hernández, J. L., Contreras-Rivera, S., Estrada-Ríos, L. A., & Machado-Encinas, G. (2024). Assessing the use of GitHub Copilot on students of engineering of information systems. *International Journal of Software Engineering and Knowledge Engineering*, 34(11), 1717–1734.
- Dikici, S., & Bilgin, T. T. (2025). Advancements in automated program repair: A comprehensive review. *Knowledge and Information Systems*, 67, 4737–4783. <https://doi.org/10.1007/s10115-025-02383-9>
- Ehsani, R., Pathak, S., Parra, E., Haiduc, S., & Chatterjee, P. (2026). What characteristics make ChatGPT effective for software issue resolution? An empirical study of task, project, and conversational signals in GitHub issues. *Empirical Software Engineering*, 31 (22)
- Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., & Lahiri, S. K. (2024). LLM-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9), 2254–2268. <https://doi.org/10.1109/TSE.2024.3428972>
- Gheventer, A., Gurgel, P. do A., Brito, C. H., de Mello, R. M., Rocha, S., Feitosa, R., & Travassos, G. H. (2026). Generative AI solutions for software quality: Assessing industrial readiness. *Software Quality Journal*, 34
- Gülmez, B. (2026). Code generation with large language models: A survey from neural program synthesis to autonomous software development. *Applied Intelligence*, 56, <https://doi.org/10.1007/s10489-026-07230-0>

- Haldar, S., Pierce, M., & Capretz, L. F. (2025). 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*, 13, 46070–46090
- Hemmat, A., Sharbaf, M., Kolahdouz-Rahimi, S., Lano, K., & Tehrani, S. Y. (2025). Research directions for using LLM in software requirement engineering: A systematic review. *Frontiers in Computer Science*, 7
- Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), Article 220, 1–79. <https://doi.org/10.1145/3695988>
- Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2026). A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2), Article 58, 1–72. <https://doi.org/10.1145/3747588>
- Jiang, X., Dong, Y., Wang, L., Zheng, F., Shang, Q., Li, G., Jin, Z., & Jiao, W. (2024). Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7)
- Kande, R., Pearce, H., Tan, B., Dolan-Gavitt, B., Thakur, S., Karri, R., & Rajendran, J. (2024). (Security) assertions by large language models. *IEEE Transactions on Information Forensics and Security*, 19, 4374–4389. <https://doi.org/10.1109/TIFS.2024.3372809>
- Kang, S., Yoon, J., Askarbekkyzy, N., & Yoo, S. (2024). Evaluating diverse large language models for automatic and general bug reproduction. *IEEE Transactions on Software Engineering*, 50(10), 2677–2694.
- Kang, S., Yoon, J., Yoo, S., & Zeller, A. (2025). Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30, Article 46. <https://doi.org/10.1007/s10664-024-10594-x>
- Lee, J., Jung, W., & Baek, S. (2024). In-house knowledge management using a large language model: Focusing on technical specification documents review. *Applied Sciences*, 14(5)
- Luitel, D., Hassani, S., & Sabetzadeh, M. (2024). Improving requirements completeness: Automated assistance through large language models. *Requirements Engineering*, 29(1), 73–95.
- Lyu, M. R., Ray, B., Roychoudhury, A., Tan, S. H., & Thongtanunam, P. (2025). Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology*, 34(5), 1–33. <https://doi.org/10.1145/3708519>
- Najmi, A., & El-Dosuky, M. (2025). Intelligent software testing for test case analysis framework using ChatGPT with natural language processing and deep learning integration. *Journal of Computer Science*, 21(5), 1140–1155

- Necula, S.-C., Dumitriu, F., & Greavu-Şerban, V. (2024). A systematic literature review on using natural language processing in software requirements engineering. *Electronics*, 13(11)
- Newcomb, L., Newcomb, A., & Ochoa, O. (2025). Preconditions and postconditions as design constraints for LLM code generation. *IEEE Access*, 13, 184259–184274. <https://doi.org/10.1109/ACCESS.2025.3625819>
- Nyamsuren, E. (2025). Evaluating quantized large language models for code generation on low-resource language benchmarks. *Journal of Computer Languages*, 84, Article 101351. <https://doi.org/10.1016/j.col.2025.101351>
- Ogenrwot, D., & Businge, J. (2026). PatchTrack: A comprehensive analysis of ChatGPT's influence on pull request outcomes. *Empirical Software Engineering*, 31
- Pandey, S. K., Chand, S., Horkoff, J., Staron, M., Ochodek, M., & Durisic, D. (2025). Design pattern recognition: A study of large language models. *Empirical Software Engineering*, 30, Article 69. <https://doi.org/10.1007/s10664-025-10625-1>
- Patcas, R., Patcas, L. M., Mayer, P., & Scanniello, G. (2026). An evaluation study of large language models for addressing code issues. *Empirical Software Engineering*, 31. <https://doi.org/10.1007/s10664-026-10858-8>
- Pendyala, V. S., & Thakur, N. B. (2025). Performance and interpretability analysis of code generation large language models. *Neurocomputing*, 656, <https://doi.org/10.1016/j.neucom.2025.131461>
- Pinckney, N., Batten, C., Liu, M., Ren, H., & Khailany, B. (2025). Revisiting VerilogEval: A year of improvements in large-language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems*, 30(6), <https://doi.org/10.1145/3718088>
- Qu, Y., Huang, S., Li, Y., Bai, T., Chen, X., Wang, X., Li, L., & Yao, Y. (2025). BadCodePrompt: Backdoor attacks against prompt engineering of large language models for code generation. *Automated Software Engineering*, 32(1), <https://doi.org/10.1007/s10515-024-00485-2>
- Radliński, Ł., & Swacha, J. (2025). Large language models for early-stage software project estimation: A systematic mapping study. *Applied Sciences*, 15(24)
- Russo, D. (2024). Navigating the complexity of generative AI adoption in software engineering. *ACM Transactions on Software Engineering and Methodology*, 33(5), 1–50. <https://doi.org/10.1145/3652154>
- Sajadi, A., Rahman, A., Shar, L. K., & Poshvanyk, D. (2025). Do LLMs consider security? An empirical study on responses to

- programming questions. *Empirical Software Engineering*, 30, Article 113. <https://doi.org/10.1007/s10664-025-10658-6>
- Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1), 85–105. <https://doi.org/10.1109/TSE.2023.3334955>
- Tasarsu, M., Yigit, T., & Güngör, V. C. (2026). Test case generation using large language models: A systematic literature review. *Cluster Computing*, 29, <https://doi.org/10.1007/s10586-026-06021-z>
- Terragni, V., Vella, A., Roop, P., & Blincoe, K. (2025). The future of AI-driven software engineering. *ACM Transactions on Software Engineering and Methodology*, 34(5), 1–20. <https://doi.org/10.1145/3715003>
- Venkatesh, A. P. S., Sunil, R., Sabu, S., Mir, A. M., Reis, S., & Bodden, E. (2025). An empirical study of large language models for type and call graph analysis in Python and JavaScript. *Empirical Software Engineering*, 30, <https://doi.org/10.1109/TSE.2024.3368208>
- Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4), 911–936.
- Weber, T., Brandmaier, M., Schmidt, A., & Mayer, S. (2024). Significant productivity gains through programming with large language models. *Proceedings of the ACM on Human-Computer Interaction*, 8(EICS), 256, 1–29.
- Weyssow, M., Zhou, X., Kim, K., Lo, D., & Sahraoui, H. A. (2025). Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 34(7), <https://doi.org/10.1145/3714461>
- Wong, M.-F., Guo, S., Hang, C.-N., Ho, S.-W., & Tan, C.-W. (2023). Natural language generation and understanding of big code for AI-assisted programming: A review. *Entropy*, 25(6), Article 888. <https://doi.org/10.3390/e25060888>
- Zhang, J., Cambronero, J. P., Gulwani, S., Le, V., Piskac, R., Soares, G., & Verbruggen, G. (2024). PyDex: Repairing bugs in introductory Python assignments using LLMs. *Proceedings of the ACM on Programming Languages*, 1100–1124. <https://doi.org/10.1145/3649850>
- Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2024). Measuring GitHub Copilot’s impact on productivity. *Communications of the ACM*, 67(3), 54–63.