International Journal of Scientific Research in Computer Science and Engineering
Vol.**13**, Issue.**3**, pp.**95-103**, June **2025**
E-ISSN: 2320-7639
Available online at: www.isroset.org

Review Article

A Comprehensive Review of AI-Driven Software Engineering: Challenges, Opportunities, and Future Directions

Medhunhashini D R[1*], K S Jeen Marseline[2], Ramya U [3]

[1,2,3]Dept. of IT & Cognitive Systems, Sri Krishna Arts and Science College, Coimbatore, India

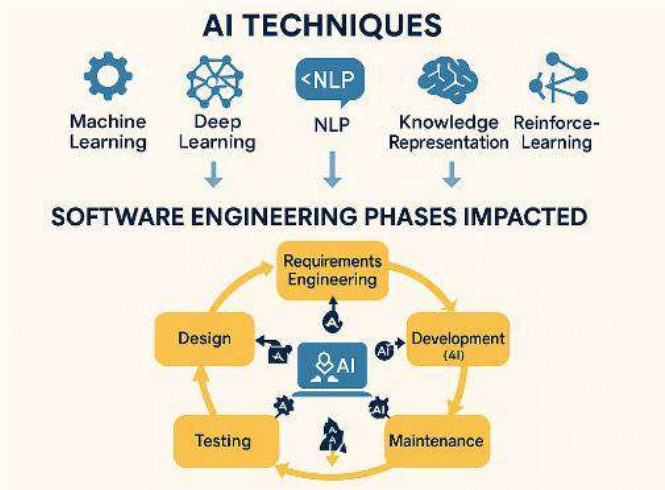
**Corresponding Author:* ✉

Received: 22/Apr/2025; **Accepted**: 23/May/2025; **Published**: 30/Jun/2025. | **DOI**: https://doi.org/10.26438/ijsrcse.v13i3.692

Abstract— The integration of Artificial Intelligence (AI) into Software Engineering (SE) has significantly transformed the landscape of software development, offering potential for enhanced efficiency, automation, and innovation across various stages of the software lifecycle. This review explores the current state of AI-driven software engineering, focusing on the advancements made from 2020 to 2025. We categorize state-of-the-art research into key application areas, including code generation, bug prediction, test case generation, software maintenance, and project management, highlighting AI's impact in automating routine tasks, improving code quality, and assisting developers in decision-making processes.AI tools such as GitHub Copilot and Codex are revolutionizing code generation by leveraging large language models to produce code snippets, entire functions, and even full programs, reducing the burden on developers. In addition, AI-driven bug prediction models are aiding developers in identifying potential issues earlier, improving defect detection and prioritization. Test case generation tools like EvoSuite and Diffblue Cover automate unit test creation, enhancing testing efficiency and ensuring better code coverage. AI also contributes to software maintenance by suggesting improvements and optimizations, thereby improving long-term code quality and sustainability. Furthermore, AI is being used in project management for sprint planning, risk prediction, and resource allocation. Despite these advancements, challenges such as explainability, data quality, tool integration, and ethical concerns remain significant. This review discusses these challenges and proposes future research directions, including human-in-the-loop systems and hybrid approaches combining symbolic reasoning with neural models. We also emphasize the need for continuous research to ensure AI becoming a reliable, ethical, and effective partner in software engineering.

Keywords— *Software Engineering, Artificial Intelligence, Bug Prediction, Ethical Concerns, Machine Learning*

Graphical Abstract



1. Introduction

The field of software engineering is undergoing a transformative shift with the integration of Artificial Intelligence (AI) technologies. AI-driven tools and methodologies are increasingly being adopted to enhance various aspects of the software development lifecycle, including code generation, testing, maintenance, and project management[1]. This evolution is not only reshaping traditional development practices but also redefining the roles of software engineers in the AI era. Recent advancements have led to the emergence of AI-powered code assistants, such as GitHub Copilot, Amazon CodeWhisperer, and Windsurf, which leverage large language models (LLMs) to assist developers in writing code more efficiently[2]. These tools have demonstrated significant potential in improving productivity and reducing the time required for software development tasks. For instance, Microsoft reported that up to

30% of its code is now written by AI, highlighting the growing reliance on AI in software engineering processes. Despite these advancements, the integration of AI into software engineering presents several challenges. Concerns regarding the explainability of AI-generated code, potential biases in AI algorithms, and the ethical implications of AI-driven development are increasingly coming to the forefront. Moreover, the rapid adoption of AI tools necessitates a re-evaluation of existing software engineering practices to ensure they align with the evolving technological landscape. This review paper aims to provide a comprehensive overview of the current state of AI-driven software engineering. It will explore the various applications of AI in software development, analyze the challenges associated with its integration, and discuss potential future directions for research and practice. By examining both the opportunities and obstacles presented by AI, this paper seeks to offer insights into how software engineering can adapt to and thrive in an AI-enhanced environment.

1.1 Overview of Software Engineering

Software engineering (SE) is a discipline that encompasses the systematic application of engineering principles to the development, maintenance, and management of software systems. It covers various activities including requirements analysis, design, implementation, testing, deployment, and evolution. Over the decades, SE has evolved to meet the increasing complexity and scale of software systems, embracing new paradigms such as agile methodologies, DevOps, and continuous integration/continuous deployment (CI/CD). Despite these advancements, traditional SE processes remain labor-intensive and often susceptible to human error, inefficiency, and inconsistencies[3].

1.2 Rise of AI and Machine Learning in Software Engineering

In recent years, the advent of Artificial Intelligence (AI) and Machine Learning (ML) has triggered a paradigm shift in software engineering. AI-powered tools are now capable of automating critical SE tasks such as code generation, defect prediction, test automation, and maintenance. For example, large language models like OpenAI Codex and Code Llama have demonstrated the ability to write syntactically correct and contextually relevant code snippets from natural language prompts[4]. Similarly, ML algorithms are increasingly used to predict bugs, prioritize tests, estimate project risks, and recommend code improvements. This integration of AI into SE processes is leading to the emergence of a new domain often referred to as AI-driven software engineering.

1.3 Motivation for the Review

The integration of AI into SE offers significant promise, it also introduces new challenges and complexities. There are growing concerns around trustworthiness, explainability, data privacy, and over-reliance on black-box models. The SE community faces a fragmented landscape where tools, datasets, and methodologies are rapidly evolving but often lack standardization and interoperability. There is a pressing need to understand the state of the art, identify gaps in current research, and explore potential pathways for the future. This review is motivated by the need to provide a structured and critical examination of how AI is reshaping SE and what it means for practitioners and researchers moving forward.

1.4 Scope and Contributions of the Paper

This review paper examines the diverse applications of Artificial Intelligence (AI) throughout the software engineering (SE) lifecycle, including code generation, testing and quality assurance, defect prediction, software maintenance, and project management. By surveying research contributions from 2020 to 2025, the paper highlights both academic advancements and industrial tools that have contributed to the field. The key contributions of this paper include a comprehensive categorization of AI-driven techniques applied to various SE activities, offering a clear overview of the current state of AI in software engineering. Additionally, a comparative analysis of the leading tools and methods is provided, emphasizing their strengths and limitations to give a balanced perspective on their effectiveness. The paper also addresses the emerging challenges in AI for SE, such as the lack of explainability, ethical concerns regarding data usage and bias, and the difficulties in integrating AI tools with traditional development workflows. Finally, the paper identifies open research questions and proposes future directions, aiming to guide the continued evolution of AI-enhanced software engineering practices.

2. Literature Review

The incorporation of AI into Engineering process has sparked significant interest in both academia and industry, driven by the promise of automation, scalability, and efficiency. The below literature review shows some prominent discussion on the AI into SE. [5]conducted a comprehensive survey on data collection and labeling challenges in SE. They emphasized the limited availability of high-quality labeled datasets and the pervasive issues of data inconsistency and noise, which hinder the machine learning model's training and testing.[6] systematically reviewed explainable AI (XAI) methods tailored to software engineering. Their findings indicate that developers often distrust black-box AI tools due to the lack of transparency in decision-making, which is especially problematic in safety-critical systems.

[7] highlighted the hallucination problem in code generation and bug prediction models, where syntactically correct but semantically flawed code is generated. They propose validation methods such as static analysis and automated testing to mitigate these risks[8] investigated the integration challenges of AI tools with conventional SE environments. Their study points to compatibility issues, the inertia of legacy systems, and the need for low-friction deployment mechanisms.[9] reviewed the legal and ethical ramifications of AI in SE. Key concerns include copyright violations from training data, lack of attribution, and inherent biases within models that can impact fairness and trust. [10] presented a case study on automating bug prediction using labeled data. They demonstrated how ML models trained on

bug-labeled repositories can enhance bug detection accuracy and reduce manual inspection efforts[11] explored techniques for making AI decisions interpretable in SE tasks. They advocated for approaches like attention maps and rule extraction to bridge the gap between model output and developer understanding. [12] discussed transfer learning's potential in code quality prediction. By fine-tuning models like BERT on domain-specific repositories, they showcased significant performance gains even with limited labeled data. [13] surveyed reinforcement learning (RL) applications in software testing. RL has shown promise in optimizing test case execution and automated code repair, though defining reward functions and ensuring convergence remain challenges. [14] examined ethical issues in AI-assisted code generation. Their paper stressed the importance of transparent attribution, license compliance, and mitigation of encoded biases to ensure responsible AI deployment in SE contexts. These studies collectively illuminate the complex landscape of applying AI in SE. They underscore the necessity for robust datasets, interpretable models, seamless tool integration, and a solid ethical foundation to support the sustainable adoption of AI technologies in software development.

3. Key Application Areas

The integration of AI into software engineering has led to significant innovations in different SDLC life cycle stages of software engineering. This section discusses five key application areas where AI technologies are making the most impact: code generation, bug prediction, test case generation, software maintenance, and project management.

3.1 AI for Code Generation

AI-powered code generation is one of the most transformative innovations in recent software engineering. Tools like GitHub Copilot**,** powered by OpenAI Codex, and Amazon CodeWhisperer utilize large language models (LLMs) trained on billions of lines of code to generate context-aware code suggestions in real time[14].These tools function as intelligent code assistants, helping developers write functions, complete boilerplate code, and even suggest entire class structures based on natural language comments. Initial evaluations suggest that Copilot can improve developer productivity, with studies indicating that developers using AI code assistants complete tasks faster and with fewer errors.AI-driven tools for software engineering present several challenges that need to be addressed for their widespread adoption. One key issue is that generated code may inadvertently introduce subtle bugs or security vulnerabilities, which could go unnoticed during automated processes. This poses significant risks, especially in mission-critical applications where precision and reliability are essential. Additionally, AI models trained on large datasets may inadvertently replicate licensed or vulnerable code, potentially leading to legal concerns or the unintentional inclusion of insecure practices. Another challenge is the lack of explainability in the generated code, as AI systems often operate as "black boxes."[15] This lack of transparency can reduce developer trust in AI-generated solutions, making it difficult for them to understand the rationale behind suggestions and verify their correctness. As a result, developers may hesitate to fully rely on AI tools for critical tasks, especially when dealing with complex or sensitive software projects..

3.2 Bug Prediction and Localization

Machine Learning (ML) techniques have become instrumental in predicting defect-prone code modules and localizing bugs, significantly enhancing software quality assurance efforts. These models leverage static code metrics, such as cyclomatic complexity and churn rate, alongside historical defect data to train classifiers like decision trees, support vector machines (SVM), and neural networks. Popular tools and methods in this domain are listed in Table 3. It includes DeepLoc, which employs deep learning algorithms to locate bugs with greater precision at a granular code level, and BugLocator, which combines information retrieval techniques and code history to effectively identify fault locations[16].

Another prominent approach is ML4SE, a framework that integrates ML pipelines into the Continuous Integration/Continuous Deployment (CI/CD) process, automating defect prediction and bug localization as part of the software delivery lifecycle. The impact of these tools is significant: by identifying vulnerable areas early, they enable developers to keep attention on their testing and code back walk efforts on the most critical parts of the code, thereby reducing the time spent on debugging and rework. This leads to faster development cycles and more reliable software, ultimately improving both code quality and productivity.

Table 1. Bug Prediction Tools

Tool/ Method	Approach	Key Features	Strengths	Limitations
DeepLoc	Deep learning-based bug localization	Uses deep learning models to pinpoint bugs at a fine-grained code level	High precision in identifying bugs, fine-grained localizatior	Requires large labeled datasets for training, computationally intensive
BugLocator	Information retrieval + code history	Combines historical defect data and information retrieval for fault localization	Efficient in leveraging code history, adaptable t various codebases	May struggle with codebases lacking historical data or complex patterns
ML4SE	Machine learning pipeline integration in CI/CD	Integrates ML models within CI/CD processes to automate defect prediction	Seamless integration with CI/CD workflows, continuous defect prediction	Limited by the quality and scope of training data in CI/CD pipelines

3.3 Test Case Generation

AI is playing a transformative role in enhancing test automation, particularly through the generation of automated test cases. AI models are now capable of parsing natural language requirements, user stories, or even UI interactions to automatically generate corresponding test scripts using popular tools like Selenium or Appium. This advancement streamlines the testing process and significantly reduces manual effort. Notable examples of AI-driven test automation tools include Diffblue Cover, which uses symbolic AI to automatically generate unit tests for Java code, and EvoSuite, which leverages evolutionary algorithms to create high-coverage JUnit tests[17]. These tools not only save valuable time by automating the creation of test cases but also ensure greater test coverage, allowing for early detection of bugs that might otherwise be missed in manual testing. Additionally, AI-driven test automation enhances continuous testing, particularly in agile workflows, by enabling faster feedback cycles, improving code quality, and ensuring that new features are properly tested throughout the development process. This results in more efficient software development, with improved test reliability and quicker identification of issues.

3.4 Software Maintenance and Refactoring

AI tools are increasingly being utilized in the maintenance and improvement of legacy systems, offering significant advantages in terms of code quality and efficiency. Machine learning-based tools can automatically suggest opportunities for code refactoring, detect code smells, and even recommend security patches, helping to keep legacy systems robust and up-to-date. Key tools in this domain include Facebook Aroma, a code-to-code search engine that identifies reuse opportunities within existing codebases, and Refactory.AI, which suggests restructuring options for complex or duplicated code, making it easier to maintain and scale. Additionally, AI-based linters can detect performance bottlenecks and recommend optimization patterns, contributing to enhanced system performance. However, despite these benefits, there are challenges. One limitation is the context sensitivity of AI tools, as they may not fully understand the specific requirements or constraints of the legacy system, potentially leading to less accurate suggestions. Furthermore, there is a risk of introducing regressions during automated refactoring, as AI-driven changes may inadvertently break existing functionality or create new issues that were not anticipated. As a result, while AI tools can be valuable in legacy system maintenance, careful consideration and manual validation remain essential to ensure the reliability of the refactorings.

3.5 Project Management

AI is increasingly being leveraged in software project management, particularly in automating critical activities such as planning, effort estimation, and risk assessment. By analyzing historical project data, AI models can accurately forecast delivery timelines, estimate the required effort, and optimize resource utilization. Key applications of AI in project management include effort estimation models that use regression and time-series forecasting to predict the time and resources needed for various tasks, improving the accuracy of project planning. Additionally, sprint planning assistants powered by natural language processing (NLP) analyze issue trackers and recommend optimal workloads for development teams based on project requirements, team capacity, and past performance. AI is also being used for risk prediction, where patterns in delay history and developer activity logs are analyzed to forecast potential bottlenecks or risks in the project, allowing proactive mitigation strategies[18]. The benefits of AI in project management are significant: it enables data-driven decision-making, where predictions and recommendations are based on real project data rather than gut instinct or guesswork. AI also helps reduce project overruns by providing more accurate timelines and effort estimations, leading to fewer unexpected delays. Moreover, by analyzing team performance and resource utilization, AI enhances resource allocation, ensuring that teams are working efficiently and effectively, ultimately leading to more successful project outcomes.

4. Emerging Techniques

AI adoption deepens in software engineering, newer and more sophisticated techniques are emerging. Among the most influential are Large Language Models (LLMs), transfer learning, and reinforcement learning, which are pushing the boundaries of automation, code understanding, and adaptive decision-making.

4.1 Use of Large Language Models (LLMs)

Large Language Models (LLMs) such as GPT-4, Code Llama, and PaLM-Coder are transforming the landscape of software engineering automation. These models, trained on vast amounts of source code and documentation, possess the ability to understand programming languages, developer intent, and natural language instructions with a level of sophistication previously unattainable.

Their capabilities are pushing the boundaries of what can be automated in software development, offering applications that enhance productivity and ease of development. For example, code completion and synthesis powered by GPT-based models can generate functions, documentation, and even entire applications based on minimal input from developers, drastically reducing development time. Additionally, conversational debugging allows LLMs to assist developers interactively by explaining errors, suggesting fixes, or even refactoring code in real time, enhancing the debugging process. Requirements analysis is another area where LLMs excel; they can extract structured specifications from unstructured user inputs or legacy documents, streamlining the process of gathering and understanding project requirements.

Despite their impressive capabilities, there are several limitations to consider. LLMs are prone to hallucinations, meaning they can generate code with syntactic or semantic errors that may not be immediately apparent. Additionally, their high computational cost and resource requirements make them challenging to implement in resource-constrained

environments. Moreover, there are significant legal and ethical concerns surrounding the training data used to develop these models, in regard to copyright, data confidentiality, and the potential for biases in the generated code. These limitations must be addressed to ensure that LLMs can be effectively and responsibly integrated into software engineering workflows[19].

4.2 Transfer Learning and Fine-Tuning for SE Tasks

Transfer learning has become a powerful technique in software engineering, particularly for tasks where high-quality labeled datasets are scarce. By allowing pre-trained models to adapt to new tasks with minimal additional data, transfer learning addresses one of the major challenges in software engineering— the lack of comprehensive, domain-specific datasets. This approach is especially beneficial for tasks like code summarization, bug detection, and software maintenance. For instance, fine-tuning a general-purpose LLM like GPT or BERT on domain-specific data from software repositories (such as GitHub or Stack Overflow) allows the model to better understand software-related language and concepts without requiring a vast amount of labeled data. Additionally, specialized models like CodeBERT and GraphCodeBERT are already pre-trained on large codebases and can be fine-tuned for tasks like code summarization, clone detection, and code search, offering substantial improvements in performance for niche software engineering tasks[20].

The key benefits of transfer learning in software engineering include a significant reduction in the need for task-specific model training from scratch, making it much more resource-efficient. Moreover, transfer learning has shown to improve performance on specialized software engineering tasks, such as predicting CI failures or identifying technical debt, by leveraging pre-existing knowledge learned from large datasets. Finally, transfer learning supports cross-language code understanding and translation, enabling models to bridge gaps between different programming languages, improving their versatility and applicability in multi-language software environments. As a result, transfer learning has proven to be a valuable tool in enhancing the automation and accuracy of software engineering processes, particularly in resource-constrained settings.

4.3 Reinforcement Learning in Software Automation

Reinforcement Learning (RL) is making significant strides in software engineering (SE) automation, particularly in scenarios that involve iterative decision-making and feedback-driven processes. In SE, RL agents learn to optimize tasks through trial-and-error interactions with their environments, such as compilers, test systems, or development platforms. These agents are designed to take actions that maximize long-term rewards based on feedback from the environment, making RL a powerful tool for automating and improving various software engineering processes. Some prominent use cases of RL in software automation include test case prioritization, where RL agents learn to identify and execute the most fault-revealing tests first, helping to detect defects early in the development cycle. Another application is automated code repair, where RL agents are rewarded for generating compliable and correct patches, offering a potential solution for bug fixing with minimal human intervention. Additionally, software optimization is another area where RL excels; agents can optimize code performance or energy efficiency by adjusting compiler flags or restructuring loops to find the most efficient configuration[21].

There are several challenges that need to be addressed for RL to reach its full potential in software engineering. One key challenge is defining meaningful reward functions that align with the goals of the task and provide useful feedback to the agent. Additionally, there is a need to balance exploration and exploitation in large and complex code spaces, where the agent must explore different actions to find optimal solutions without getting stuck in suboptimal local decisions. Finally, slow convergence in real-world environments remains a challenge, as RL agents often require significant amounts of training time and computational resources to achieve satisfactory results, particularly in complex software systems with numerous variables.

Despite these challenges, the use of RL in software automation holds great promise, offering the potential to improve efficiency, accuracy, and innovation in software development.

5. Challenges in AI Models

5.1 Data Availability and Labeling

AI models especially those based on deep learning, will be in need for big data to ensure smooth and validated training. In software engineering, however, such datasets are often difficult to obtain. Many datasets are scattered across various repositories like GitHub or Bitbucket, which means data needs to be manually aggregated from different sources. Moreover, the quality of the data can vary significantly; for example, commit messages may be noisy or ambiguous, and issue trackers can be incomplete or inaccurately labeled. This inconsistency complicates the creation of reliable datasets for training AI models. Additionally, for tasks like bug-fix localization or intent classification, labeled data can be particularly sparse, making it hard to train models effectively. Manual labeling of software artifacts, such as tagging defects or annotating code behavior, is a time-consuming and domain-specific task that limits the scalability of supervised learning approaches. These challenges underscore the need for innovative data collection and labeling strategies to enable effective AI deployment in software engineering.

5.2 Explainability and Trustworthiness

One of the most significant barriers to the adoption of AI in software engineering is the lack of explainability in the decision-making process of AI models. Developers are often hesitant to trust AI-generated code, bug predictions, or test cases if the rationale behind the decisions is not transparent. Many black-box models provide little to no traceability, making it difficult for developers to debug or validate AI suggestions. This lack of interpretability can be particularly

problematic in safety-critical systems, such as those used in aerospace or finance, where regulatory standards demand explainable decision-making. Without clear explanations for how AI models arrive at their conclusions, developers are less likely to fully trust AI-generated outputs, which can hinder widespread adoption. To overcome this challenge, researchers are exploring techniques such as attention visualization, rule extraction, and the integration of symbolic and neural models to enhance model transparency and improve trust in AI-generated results.

5.3 Model Hallucination and Inaccuracies

Generative models, such as Large Language Models (LLMs), are often prone to hallucination—a phenomenon where the model generates outputs that are syntactically correct but semantically incorrect or misleading. In the context of code generation, this can lead to generated code that fails to compile, contains security vulnerabilities, or violates coding standards. This issue of hallucination can also affect bug prediction, where the model may raise false positives, diverting developer attention to non-issues and diminishing the tool's credibility. These inaccuracies can significantly reduce the reliability of AI-based software engineering tools, especially in environments that demand high precision and correctness. To mitigate these risks, robust validation techniques such as unit testing, static analysis, and manual reviews are essential. These methods can help ensure that the generated code meets quality standards and doesn't introduce errors that could compromise the software's integrity or security.

5.4 Integration with Traditional SE Tools and Workflows

Another challenge to the widespread adoption of AI in software engineering is the difficulty of integrating AI tools into traditional development workflows. Many AI systems are designed to operate as standalone tools, lacking plug-and-play compatibility with existing Integrated Development Environments (IDEs), Continuous Integration/Continuous Deployment (CI/CD) pipelines, or version control systems. Teams working with legacy or custom-built systems may find it particularly difficult to incorporate modern AI components into their established workflows, leading to increased complexity and overhead. Additionally, software development processes are often dynamic, with codebases evolving continuously. AI models must be able to adapt in near real-time to these changes, which presents a challenge in incremental and continual learning. Seamless integration of AI into familiar tools is critical for ensuring that developers can use AI systems without significant disruption to their usual processes. Achieving this integration will be crucial for facilitating widespread adoption of AI in software engineering.

5.5 Ethical and Legal Concerns

The growing use of AI in software engineering raises significant ethical and legal concerns, particularly regarding copyright and intellectual property. For example, AI models trained on free available repositories may unintentionally reproduce copyrighted code, violating licensing agreements and creating potential legal liabilities. Another concern is attribution—AI-generated code may not properly credit the original authors, leading to ambiguity around authorship and intellectual property rights. Additionally, AI models may encode historical biases found in training data, which can result in unfair or discriminatory outcomes. This could affect areas such as code review comments or even hiring recommendations, where biased models may perpetuate stereotypes or exclude underrepresented groups. To address these concerns, it is important to improve transparency in the datasets used to train AI models, ensure stricter compliance with licensing agreements, and develop ethical standards for the responsible use of AI in software engineering. This would help mitigate risks associated with bias, copyright infringement, and attribution, ensuring that AI's integration into SE is both legally sound and ethically responsible[22].

6. Results and Discussions

6.1 Summary of Leading Tools and Research Projects

A wide variety of AI-powered tools and research projects have been developed to support software engineering tasks. In the area of code generation, tools like GitHub Copilot, Amazon Code Whisperer, and Code Llama leverage large language models to generate code from natural language prompts. For bug prediction and localization, models such as BugLocator, DeepLoc, and various ML-based classifiers have shown strong potential in identifying vulnerable modules. Test case generation has benefited from tools like Diffblue Cover and EvoSuite, which employ symbolic AI and evolutionary algorithms, respectively, to automate the creation of unit tests. Academic models like CodeBERT, GraphCodeBERT, and PLBART support a range of SE tasks, including code summarization, clone detection, and code translation. Each of these tools serves a specific niche in the SE lifecycle, reflecting the increasing specialization within the AI-for-SE community.

6.2 Metrics Used for Evaluation

Evaluation of AI tools in software engineering depends heavily on the targeted task and the context in which the tool is deployed. Common metrics include accuracy, precision, recall, and F1-score for classification-based models, especially in bug prediction and static analysis. In code generation, performance is often assessed using BLEU scores, edit distance, or developer satisfaction surveys that rate code readability and correctness. For search and recommendation systems, metrics like Mean Reciprocal Rank (MRR), NDCG, and top-k accuracy are widely used. Test case generation tools are evaluated based on code coverage (e.g., statement, branch, or mutation coverage) and fault detection rate. However, a challenge in this domain is the lack of standardized benchmarks and datasets, which makes fair comparison across tools and studies difficult.

6.3 Performance and Limitations

While the reported performance of AI-based tools in software engineering is promising, many limitations remain. Most tools show high accuracy on curated or open-source datasets but struggle when applied to proprietary, complex, or domain-specific codebases. Generalization remains a key

concern, especially for large language models trained on public data. Moreover, several tools depend on high-quality labeled datasets, which are often unavailable in industrial settings. In terms of usability, integration with traditional development environments (e.g., IDEs, CI/CD pipelines) is still in its early stages, reducing their immediate impact in production workflows. Hallucination in code generation, poor explainability in defect prediction, and high computational demands in training or inference are also notable challenges. Despite these issues, AI tools continue to mature, and their practical value increases as more reliable models, APIs, and datasets become available.

7. Open Research Problems

Despite the significant progress made in integrating AI into software engineering, several research challenges remain. These open problems highlight areas that require further exploration to unlock the full potential of AI tools in real-world development environments.

7.1 Ensuring Reliability in AI-Generated Code

A pressing challenge is ensuring the steadfastness of AI-generated code. While tools like GitHub Copilot and Amazon CodeWhisperer are capable of producing syntactically correct code, ensuring that this code is functionally correct, secure, and optimized remains a complex task. AI-generated code often lacks the robustness of manually written code, potentially introducing subtle bugs or security vulnerabilities. Developing methods for automated verification**,** formal methods, and dynamic analysis to validate AI-generated code is critical. Furthermore, integrating unit tests and static analysis within the code generation process can help catch errors earlier and improve reliability.

7.2 Real-Time Collaboration Between AI and Developers

AI-powered tools that assist in code generation, debugging, or testing must evolve to enable real-time collaboration between AI systems and developers. Currently, most AI tools function in a reactive mode, where they provide suggestions after code has been written or errors have been encountered. However, to achieve true collaborative intelligence, AI tools need to operate proactively within the development flow. For instance, tools that offer context-aware suggestions or real-time error detection during code composition can significantly improve developer productivity. Research into developing more interactive and responsive AI systems that can engage developers in a collaborative, seamless manner is an important area of focus.

7.3 Continuous Learning from Evolving Codebases

Software systems evolve continuously, with new features, bug fixes, and refactoring constantly changing the underlying codebase. AI models trained on static datasets often fail to keep up with these ongoing changes. One open research problem is the development of continuous learning frameworks that allow AI systems to adapt to changes in evolving codebases without requiring retraining from scratch. Such systems would need to efficiently process new code while preserving learned knowledge and avoiding catastrophic forgetting. Additionally, online learning and incremental training methods must be explored to make AI systems more adaptable and scalable in real-world software engineering environments.

7.4 Cross-Project Generalizability

AI models trained on specific projects or programming languages often struggle to generalize across different codebases, programming paradigms, or software domains. This lack of generalizability is a significant barrier to the widespread adoption of AI in software engineering. For example, a model trained to detect defects in a Java project may not perform well on a Python or C++ codebase. Addressing this problem requires the development of more robust, cross-project AI models that can transfer knowledge across different contexts and programming languages. Transfer learning methods. domain adaptation, and multi-task learning are promising approaches to achieve better cross-project performance[23].

These open research problems represent exciting opportunities for the AI community to refine existing tools and develop new methodologies that will enable AI to play a more integrated and impactful role in software engineering. Continued collaboration between academia, industry, and open-source communities will be essential to addressing these challenges and achieving the vision of fully autonomous software development.

8. Conclusion and Future Directions

The integration of AI into software engineering has made remarkable strides, offering new tools and methodologies that can automate routine tasks, improve code quality, and assist developers in various stages of the software development lifecycle. From code generation and bug prediction to test case automation and software maintenance, AI-driven approaches are enhancing productivity and enabling more efficient workflows. However, as the field continues to evolve, several challenges remain, including ensuring the reliability of AI-generated code, addressing ethical concerns, and improving the generalizability of AI models across diverse codebases. This review has explored the current landscape of AI applications in software engineering, highlighting the strengths, limitations, and emerging research directions. As AI models become more sophisticated, their role in software engineering is expected to expand further, with a focus on human-in-the-loop systems**,** hybrid symbolic-neural approaches, and the development of regulatory frameworks**.** These advancements will help to address current limitations, improve the collaboration between AI systems and developers, and ensure the ethical and legal use of AI in software development. The future of AI in software engineering holds exciting possibilities, and as the technology matures, it will be crucial to balance technical innovation with responsible practices. Through continued research, collaboration, and thoughtful regulation, AI can transform the way software is developed, tested, and maintained, offering new levels of automation, efficiency, and reliability.

AI continues to reshape the landscape of software engineering, several promising research directions are emerging. These future advancements focus on enhancing the interaction between AI and developers, improving the robustness of AI systems, and addressing the ethical and regulatory concerns that arise with AI adoption in software engineering.AI in software engineering is the development of human-in-the-loop (HITL) AI systems. HITL systems involve a continuous interaction between AI and human experts, where AI assists in automating repetitive tasks while the human developer provides oversight and corrective feedback. This approach allows for the combination of human expertise with AI's computational power, leading to more efficient workflows and improved decision-making. For instance, HITL systems could enable real-time bug detection, code generation, or testing while allowing developers to intervene and make final decisions based on context or judgment. The integration of human feedback in training AI models could also lead to more context-aware systems that understand developer preferences and project-specific constraints.

Another promising direction is the development of hybrid AI systems that combine symbolic reasoning with neural networks**.** While neural networks excel at learning patterns from large datasets, symbolic reasoning allows for logic-based problem-solving, offering transparency, explainability, and consistency. Combining these two approaches could lead to AI systems that not only generate code or predict bugs but also reason about the semantics of the code and provide explainable and verifiable solutions. For example, a hybrid system could automatically generate code while ensuring it adheres to predefined specifications and constraints. This combination could be particularly valuable in safety-critical applications where both accuracy and interpretability are paramount.

AI continues to gain prominence in software engineering, the need for regulatory and governance frameworks becomes increasingly important. These frameworks would provide guidelines on how AI systems should be used, ensuring they adhere to ethical standards, legal requirements, and privacy concerns. For example, AI tools in software engineering could be subjected to guidelines on intellectual property rights**,** data privacy**,** and bias mitigation to prevent unintended legal or ethical consequences. Establishing clear regulations for the use of AI, particularly in areas such as code generation**,** bug prediction, and software maintenance, would help to build trust among developers and stakeholders. Furthermore, regulatory frameworks could address issues related to accountability, ensuring that developers and organizations remain responsible for AI-generated outcomes.

These future directions indicate a shift towards collaborative AI systems**,** where human expertise and machine intelligence work in harmony, and towards more transparent, ethical, and regulated AI applications. As AI continues to evolve, addressing these challenges will ensure that AI-driven tools in software engineering are not only effective but also ethical, safe, and widely adopted.

Conflict of Interest

This review study has not been considered for publishing anywhere and not been disseminated. There is no conflict of interest.

Funding Source

No external funding has been availed for this study.

Author Contributions

The authors of this review paper have equally contributed for the study and further structuring with verification of the content.

Acknowledgments

We sincerely thank our management, department of IT and Cognitive Systems and the faculty team for their support in successful completion of this study.

References

[1] A. Huzzat, A. Anpalagan, A. S. Khwaja, I. Woungang, A. A. Alnoman, and A. S. Pillai, “A comprehensive review of Digital Twin technologies in smart cities,” *Digit. Eng.*, vol. 4, p. 100040, 2025, doi: https://doi.org/10.1016/j.dte.2025.100040.

[2] S. Hosseini and H. Seilani, “The role of agentic AI in shaping a smart future: A systematic review,” *Array*, vol. 26, p. 100399, 2025, doi: https://doi.org/10.1016/j.array.2025.100399.

[3] D. Ryu and J. Baik, “Effective multi-objective naïve Bayes learning for cross-project defect prediction,” *Appl. Soft Comput.*, vol. 49, pp. 1062–1077, 2016, doi: https://doi.org/10.1016/j.asoc.2016.04.009.

[4] G. Naeem, M. Asif, and M. Khalid, “Industry 4.0 digital technologies for the advancement of renewable energy: Functions, applications, potential and challenges,” *Energy Convers. Manag. X*, vol. 24, p. 100779, 2024, doi: https://doi.org/10.1016/j.ecmx.2024.100779.

[5] M. W. D. Mustafa, K. S. A. Alam, and D. S. D. S. Zawoad, “A Survey on Data Collection and Labeling in Software Engineering,” *Softw. Eng. An Int. J.*, vol. 12, no. 3, pp. 121–134, 2019.

[6] R. N. Chenthamarakshan, V. N. P. Chidambaram, and R. S. S. R. Murthy, “Explainable AI for Software Engineering: A Systematic Review,” *IEEE Access*, vol. 8, pp. 84267–84282, 2020.

[7] A. K. Smith, P. R. Thiel, and S. L. Davis, “Challenges of Model Hallucination in Code Generation and Bug Prediction BT - Proceedings of the 2021 ACM/IEEE International Conference on Software Engineering (ICSE),” 2021, pp. 104–112. doi: 10.1109/ICSE43902.2021.00024.

[8] J. R. Parsons, P. M. Rhodes, and T. A. Johnson, “Challenges in Integrating AI Systems with Traditional Software Engineering Tools,” *Softw. Syst. Model.*, vol. 20, no. 2, pp. 509–523, 2021, doi: 10.1007/s10270-020-00854-3.

[9] M. L. P. Sherman and T. K. Moore, “Legal and Ethical Implications of AI in Software Engineering,” *J. Leg. Asp. Inf. Technol.*, vol. 11, no. 4, pp. 274–286, 2020.

[10] L. Zhang, Y. H. Choi, and D. D. Lin, “Automating Bug Prediction with Data Labeling: A Case Study BT - Proceedings of the 2021 International Conference on Software Engineering (ICSE),” 2021, pp. 1120–1131. doi: 10.1109/ICSE43902.2021.00028.

[11] D. H. Liu, W. S. Yang, and F. M. Leong, “Explaining AI Decisions in Software Engineering: Techniques and Challenges,” *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, pp. 21–45, 2020, doi: 10.1145/3364699.

[12] K. S. Arif, S. P. Davis, and C. M. Tang, “Transfer Learning for Code Quality Prediction: A Comprehensive Survey,” *Empir. Softw. Eng.*, vol. 26, no. 5, pp. 987–1008, 2021, doi: 10.1007/s10664-021-09940-w.

[13] F. J. Zhang, H. S. Khan, and M. D. Ali, “Reinforcement Learning

in Automated Software Testing: An Overview," *IEEE Trans. Softw. Eng.*, vol. 47, no. 3, pp. 543–557, 2021, doi: 10.1109/TSE.2020.3018589.
[14] S. K. Gupta, H. M. Lee, and P. H. Singh, "Ethical Considerations of AI in Code Generation," *J. Softw. Ethics*, vol. 18, no. 2, pp. 45–56, 2020.
[15] Y. Wu, Z. Huang, J. Zhang, and X. Zhang, "Grouting defect detection of bridge tendon ducts using impact echo and deep learning via a two-stage strategy," *Mech. Syst. Signal Process.*, vol. 235, p. 112955, 2025, doi: https://doi.org/10.1016/j.ymssp.2025.112955.
[16] M. Nevendra and P. Singh, "TRGNet: a deep transfer learning approach for software defect prediction," *Expert Syst. Appl.*, vol. 282, p. 127799, 2025, doi: https://doi.org/10.1016/j.eswa.2025.127799.
[17] N. Limsettho, K. E. Bennin, J. W. Keung, H. Hata, and K. Matsumoto, "Cross project defect prediction using class distribution estimation and oversampling," *Inf. Softw. Technol.*, vol. 100, pp. 87–102, 2018, doi: https://doi.org/10.1016/j.infsof.2018.04.001.
[18] L. He, R. Chen, J. Hu, Z. Huang, L. Zhou, and H. Zhang, "Research on safety risk assessment model of construction engineering based on attention mechanism and graph neural network," *Syst. Soft Comput.*, vol. 7, p. 200271, 2025, doi: https://doi.org/10.1016/j.sasc.2025.200271.
[19] A. John, R. Alhajj, and J. Rokne, "A systematic review of AI as a digital twin for prostate cancer care," *Comput. Methods Programs Biomed.*, vol. 268, p. 108804, 2025, doi: https://doi.org/10.1016/j.cmpb.2025.108804.
[20] X. Ju, Y. Cao, X. Chen, L. Gong, V. Chakma, and X. Zhou, "JIT-CF: Integrating contrastive learning with feature fusion for enhanced just-in-time defect prediction," *Inf. Softw. Technol.*, vol. 182, p. 107706, 2025, doi: https://doi.org/10.1016/j.infsof.2025.107706.
[21] Y. Elomari *et al.*, "A hybrid data-driven Co-simulation approach for enhanced integrations of renewables and thermal storage in building district energy systems," *J. Build. Eng.*, vol. 104, p. 112405, 2025, doi: https://doi.org/10.1016/j.jobe.2025.112405.
[22] K. I. Gandhi and N. S. Prathyusha, "Chapter 17 - Harnessing digital twins and AI integration for enhanced disease prediction in the evolution of healthcare," in *Information Technologies in Healthcare Industry*, P. O. De Pablos, M. N. Almunawar, and M. B. T.-D. H. Anshari Digital Transformation and Citizen Empowerment in Asia-Pacific and Europe for a Healthier Society, Eds., Academic Press, 2025, pp. 361–387. doi: https://doi.org/10.1016/B978-0-443-30168-1.00004-9.
[23] S. Kanwar, L. K. Awasthi, and V. Shrivastava, "Candidate project selection in cross project defect prediction using hybrid method," *Expert Syst. Appl.*, vol. 218, p. 119625, 2023, doi: https://doi.org/10.1016/j.eswa.2023.119625.

AUTHORS PROFILE

Medhunhashini D R is currently serving as an Assistant Professor in the Department of IT & Cognitive Systems at Sri Krishna Arts and Science College. She holds M.Sc., M.Phil., and MCA degrees and is presently pursuing her Ph.D. She is a distinguished academic, having secured the 8th rank in M.Sc. Software Systems (5-Year Integrated Course) from Bharathiar University. Her credentials include being a Red Hat Certified RHCSA Trainer and a Certified Scrum Master Practitioner under Scrum Alliance. With a decade of experience, she has actively engaged in academic enrichment, having participated in around 47 Faculty Development Programmes, workshops, and training sessions. Dr. Medhun Hashini has presented and participated in over 20 national and international conferences and has published 8 research papers in reputed Scopus and Web of Science journals. She is also the author of two books.

Dr. K. S. Jeen Marseline, MCA, M.Phil., Ph.D., is a highly esteemed educator with 28 years of devoted experience in the Computer Science field. Throughout her academic journey, she has held numerous leadership positions, making significant contributions to the development of the institutions she has been a part of. She has presented over 20 research papers at both national and international conferences, resulting in 45 published research articles in SCOPUS, WoS, and peer-reviewed journals.Additionally, she has authored 7 books and contributed to 12 book chapters. She has also secured funding from the ICSSR and provided consultancy amounting to around 10 lakhs.Currently, she is the Dean of Computer Science and Mathematics at Sri Krishna Arts and Science College. In 2014, she served as the Controller of Examinations at the same college, before taking on the same position at Sri Krishna College of Engineering and Technology from 2015 to 2017.

Ms. U. Ramya, M.C.A., NET, is a Research Scholar (Part-time) at Nehru Arts and Science College, specializing in Data Mining. She currently serves as an Assistant Professor in the Department of IT & CG at Sri Krishna Arts and Science College, bringing 7.5 years of teaching experience to her role. Ms. Ramya has earned various online certifications, including Coursera courses and the RedHat Linux Certified Administrator credential. She has actively contributed to academic research, presenting papers at both national and international conferences. Additionally, she has recently authored a book titled *"Integrated AI and IoT with Self-Healing Materials for Smart Repairs."*

