Int. J. Sci. R. Tech., 2026 3(5)
A Multidisciplinary peer-reviewed Journal
www.ijsrtjournal.com [ISSN: 2394-7063]

Original Article

The Generative SDLC: A Systematic Review of Integrating Modern LLMs in Software Development Life-Cycle

A. Mohanlal, Feisal Alaswad*, Batoul Aljaddouh

Department of Computing Technologies, SRM Institute of Science and Technology, India

ABSTRACT

The rapid advancement of artificial intelligence (AI), machine learning (ML), deep learning (DL), and large language models (LLMs) has created transformative opportunities across every phase of the Software Development Life Cycle (SDLC). This paper presents a comprehensive review of the current state of AI integration in software engineering, examining each SDLC stage: requirements engineering, system design, implementation, testing, deployment, and maintenance. Particular emphasis is placed on modern LLMs such as GPT-4, Codex, and their derivatives, which have emerged as powerful tools capable of automating routine engineering tasks, augmenting developer productivity, and reshaping how software effort is estimated. Drawing upon a curated corpus of recent literature, the review synthesises theoretical frameworks and empirical findings, identifies persistent challenges including hallucination, context limitations, and bias, and outlines future research directions. The paper demonstrates that while LLMs offer significant potential as qualitative decision-support tools and AI pair programmers, their integration into safety-critical and large-scale production environments requires careful architectural alignment, cost-aware evaluation, and robust human oversight.

Keywords: Artificial Intelligence; Large Language Models; Software Development Life Cycle; Software Engineering; Machine Learning; Deep Learning; GPT; GitHub Copilot; Software Testing; Requirements Engineering.

INTRODUCTION

Artificial Intelligence (AI) has emerged as a transformative technology across numerous domains, enabling data-driven decision making, automation, and intelligent prediction. Recent advances in machine learning, deep learning, and large language models have led to significant applications in transportation systems [1], [2], financial services [3], marketing analytics [4], healthcare and medicine [5], [6], satellite and remote sensing technologies [7], cybersecurity [8], and scientific research [9]. As AI capabilities continue to evolve, their impact is extending beyond domain-specific applications to reshape the processes used to design, develop, and maintain software systems. Software engineering is a discipline grounded in systematic processes that convert human requirements into functioning, reliable systems. The Software Development Life Cycle (SDLC) provides a structured framework comprising requirements gathering, system design, implementation, testing, deployment, and maintenance. For decades, each phase has relied primarily on human expertise, domain knowledge, and manual effort. However, the proliferation of AI—spanning classical machine learning, deep neural networks, and, most recently, large language models—is reshaping this landscape in profound ways [10], [11]. The emergence of transformer-based architectures and the subsequent rise of LLMs such as OpenAI's GPT series, Meta's Code Llama, and Google's Gemini have attracted considerable attention in the software engineering community. GitHub Copilot, powered by OpenAI's Codex model, is one of the most widely deployed AI coding assistants, with studies indicating that developers using Copilot code up to 55% faster and are 73% more likely to maintain their workflow on repetitive tasks [12]. Such figures underscore the paradigm shift underway: AI is no longer merely a research curiosity but an operational reality in industrial software development. Beyond code generation, AI techniques are being employed across the entire SDLC. Natural language processing (NLP) models assist with

Relevant conflicts of interest/financial disclosures: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

eliciting, classifying, and tracing software requirements [13], [14]. ML-based classifiers predict software defects and quality metrics before release [15]. Automated test generation tools leverage LLMs to produce unit, integration, and system test cases with high semantic coverage [16]. Meanwhile, effort estimation — a perennial challenge in software project management — is being revisited through the lens of LLMs, prompting the reconceptualization of traditional cost models such as COCOMO and Function Points in LLM-assisted workflows [17], [18]. Despite remarkable progress, significant challenges persist. LLMs can generate plausible but incorrect code, a phenomenon known as hallucination. Their context windows, while expanding, remain bounded, limiting their utility on large codebases. Security, privacy, and intellectual-property concerns arise when proprietary source code is shared with cloud-hosted models. Furthermore, widely adopted estimation paradigms were designed with human labour as the primary unit of effort; their misalignment with LLM-assisted development introduces systematic biases that must be addressed [19]. This paper aims to provide a structured and evidence-based review of AI integration across all SDLC phases, with specific focus on LLMs. The remainder of the paper is organised as follows: Section 2 reviews AI in requirements engineering; Section 3 covers system design; Section 4 addresses implementation and code generation; Section 5 discusses testing and quality assurance; Section 6 examines effort estimation and project planning; Section 7 covers deployment and DevOps; Section 8 discusses maintenance; Section 9 addresses challenges and limitations; and Section 10 concludes with future directions.

Figure 1 provides an overview of how AI and large language models are integrated across the Software Development Life Cycle (SDLC), highlighting representative applications, tools, and expected outcomes throughout the requirements, design, implementation, testing, planning, deployment, and maintenance phases.

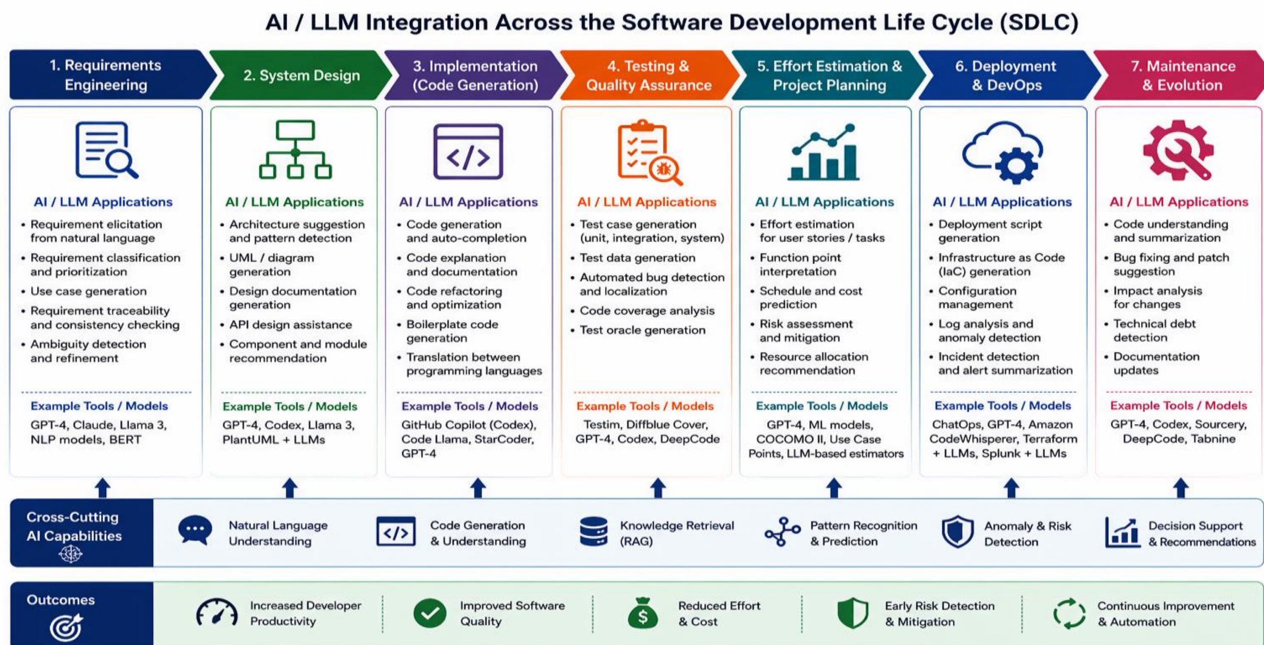

Figure 1: AI and LLM integration across the Software Development Life Cycle (SDLC).

2. AI in Requirements Engineering

Requirements engineering (RE) is widely regarded as the most consequential phase of the SDLC — defects introduced at this stage propagate and amplify throughout downstream phases. AI techniques, and LLMs in particular, are being applied to automate and augment a range of RE tasks including elicitation, classification, traceability, ambiguity detection, and specification generation [20].

2.1 Natural Language Processing for Requirements Elicitation

Traditional RE relies heavily on interviews, workshops, and document analysis. NLP approaches

automate the extraction of requirements from natural language artefacts such as user stories, emails, and meeting transcripts. Early work employing rule-based systems and classical ML models laid the groundwork; however, modern transformer-based LLMs have substantially advanced the field [13], [21]. Krishna et al. demonstrated that GPT-4 and Code Llama can produce Software Requirements Specification (SRS) documents that score on par with human benchmarks across composite metrics measuring completeness, consistency, correctness, clarity, feasibility, traceability, modularity, and compliance [19]. This suggests that, for routine SRS generation, LLMs can achieve entry-level engineer performance — a finding with significant implications for reducing time-to-specification in agile projects. Several studies have explored using LLMs for requirements elicitation interview scripting, non-functional requirements identification, and hazard analysis. In safety critical domains such as autonomous driving, LLM-driven pipelines have been reported to reduce Hazard Analysis and Risk Assessment (HARA) cycles from months to a single day, with expert teams maintaining verification oversight [19].

2.2 Requirements Classification and Traceability

Classifying requirements into functional and non-functional categories, and establishing bidirectional traceability between requirements and design artefacts, are labour-intensive tasks. ML-based classifiers, including BERT-based fine-tuned models, have demonstrated high accuracy in automated requirement classification. LLMs add a generative dimension, enabling on-demand explanation of traceability links and automated population of traceability matrices [22], [23]. Advanced prompt engineering and Retrieval Augmented Generation (RAG) strategies further enhance the accuracy of decoder-only LLMs on RE tasks, particularly when domain specific corpora are used for context retrieval. Such approaches mitigate hallucination by grounding model outputs in verified documentation [24].

2.3 Automated UML Artefact Generation

One of the most direct applications of AI in RE is the automated generation of UML artefacts from textual requirements. Transforming natural language specifications into class diagrams, use case diagrams, and sequence diagrams remains a challenging problem at the intersection of software engineering and NLP. A hybrid approach combining rule-based techniques with Naïve Bayes for named entity recognition achieved a recall of 89.44% and precision of 83.72% in generating UML class diagrams from textual requirements [25]. Building on this foundation, the PlantUCD dataset was introduced to support data-driven approaches to requirement to-model transformation. PlantUCD pairs natural language software requirements with corresponding PlantUML class diagram representations and structured abstract syntax trees, released publicly under the MIT License to facilitate reproducible research in neural code generation and model-driven development [26]. The availability of such curated datasets is critical for training and evaluating LLMs on requirement-to model tasks. More recent work using LLMs such as GPT4 for generating UML sequence diagrams from requirements shows that while models generally conform to UML standards, their correctness with respect to the specified requirements can be challenging — particularly in the presence of requirement smells such as ambiguity and inconsistency [27]. This reinforces the necessity of human expert validation in the RE loop.

3. AI in System Design

The system design phase translates verified requirements into architectural blueprints, component specifications, and data models. AI assistance in this phase is growing, ranging from architectural pattern recommendation to automated API design and design smell detection.

3.1 Architecture and Design Pattern Recommendation

LLMs are being evaluated as AI advisors for architectural decision-making. Given a set of functional and non-functional requirements, LLMs can suggest candidate architectural patterns (e.g., microservices, event-driven, layered) with supporting rationale. While early evaluations are promising, systematic benchmarking against human architects remains limited, and LLM recommendations should be treated as decision support rather than authoritative outputs [11], [13]. Multi-agent LLM frameworks represent a more structured approach to design automation. MetaGPT, for instance, encodes software

engineering practices into a collaborative multi-agent system where individual agents assume roles such as product manager, architect, and engineer, collectively generating software artefacts including requirements documents, design specifications, and code from a single high-level prompt [28].

3.2 AI-Assisted API and Interface Design

API design is a critical aspect of system design that determines long-term maintainability and interoperability. LLMs trained on large corpora of open-source APIs can suggest RESTful endpoint structures, parameter schemas, and documentation stubs. The SWE-agent framework demonstrates that LLM-based agents equipped with tool-use capabilities can interact with development environments to generate and test design artefacts autonomously [29].

4. AI in Software Implementation

The implementation phase — writing, reviewing, and debugging code — is the domain where AI-assisted tools have achieved the most visible commercial traction. LLMs for code generation represent one of the most active areas of software engineering research, with dozens of specialised models and evaluation benchmarks emerging in recent years.

4.1 Code Generation with LLMs

The release of OpenAI Codex and its integration into GitHub Copilot marked a watershed moment in AI-assisted software development. GitHub Copilot uses an LLM to provide inline code suggestions within integrated development environments, and empirical studies indicate substantial productivity gains: developers complete tasks up to 55% faster with measurably higher workflow continuity [12]. Subsequent models, including GPT-4, Code Llama, StarCoder, and DeepSeek Coder, have further expanded the frontier of automated code synthesis across multiple programming languages. Code generation LLMs have been evaluated on standardised benchmarks such as HumanEval, MBPP, and SWE-bench. Performance has improved dramatically — from approximately 50% pass rates on HumanEval in 2021 to 90%+ accuracy with modern frontier models [12]. Nevertheless, benchmarks that were once discriminating become obsolete as models improve, necessitating continuous evolution of evaluation methodologies. For code quality beyond mere correctness, the SelfEdit technique leverages execution outcomes to iteratively refine LLM-generated code through fault-aware code editing, demonstrating significant quality improvements in competitive programming benchmarks [11]. Similarly, self-collaboration frameworks that simulate software development teams using multiple LLM instances in distinct roles have shown improved problem-solving capabilities in multi-file, multi-function codebases [11].

4.2 AI-Assisted Code Review

Automated code review is a complementary application of LLMs in implementation. Systems such as CORE (Code Quality Issue Resolution) leverage LLMs to identify and resolve technical debt, code smells, and anti-patterns [29]. LLM-based code review tools can explain the rationale behind suggested changes in natural language, improving developer understanding and buy-in — a capability absent from purely static analysis tools [30].

4.3 Federated and Privacy-Preserving AI for Development

The adoption of AI during the software implementation phase raises important security and privacy concerns, particularly when sensitive development artifacts, proprietary code, or organizational data are processed by centralized AI services [31]. Federated learning offers a privacy-preserving approach by enabling multiple participants to collaboratively train machine learning models without sharing raw data. Studies on federated learning have demonstrated that distributed participants can achieve high model performance while maintaining data confidentiality [32]–[34]. In software engineering, federated learning can support the development of intelligent implementation tools by allowing organizations to collaboratively improve AI models using locally stored development data, thereby preserving privacy, protecting intellectual property, and complying with data governance requirements [35].

5. AI in Software Testing and Quality Assurance

Software testing consumes a disproportionate share of the SDLC budget, with estimates suggesting that 40–50% of total development effort is devoted to testing activities. AIdriven testing tools have the potential to reduce this burden while improving defect detection rates, test coverage, and test maintainability [36], [37].

5.1 Automated Test Case Generation

LLM-based test generation has emerged as one of the most practically impactful applications of AI in software engineering. Studies of GitHub Copilot's test generation capabilities show that LLMs can produce readable and functionally valid unit tests both within and without existing test suites, with code commenting strategies significantly influencing generation quality [16]. Tools such as TestPilot achieve 52.8% branch coverage on npm packages, improving 27% over feedbackdriven alternatives [38], [39]. LLMs bring semantic understanding to test generation that traditional search-based tools lack. While EvoSuite and similar tools rely on structural information and code execution paths, LLMs understand functional intentions and semantic boundaries, enabling the generation of tests for edge cases that structural analysis would miss. CANDOR demonstrates end-to-end Java unit test generation that surpasses EvoSuite on line coverage metrics. For integration testing, XUAT-Copilot automates mobile payment application testing, while Logi-Agent enables automated detection of business logic errors in REST APIs [38].

5.2 Software Quality Prediction

Prior to testing, predicting which software modules are likely to be defective enables targeted testing resource allocation. Machine learning techniques are the most suitable approaches for predicting software quality, with substantial research interest in predicting software reliability [40]. A comprehensive analysis of ML techniques and software metrics published between 2010 and 2021 confirms that ML-based quality prediction models provide researchers and practitioners with effective means to plan testing strategies and direct quality assurance resources [15]. Deep learning models, including convolutional and recurrent neural networks, have further advanced fault prediction accuracy on object-oriented software metrics [41].

5.3 Vulnerability Detection and Security Testing

Security testing is a specialised and increasingly critical domain where LLMs offer substantial potential [42]. LLM-based vulnerability detection systems can analyse thousands of lines of code to identify security weaknesses, with the added advantage of generating natural-language explanations of identified vulnerabilities [29], [43]. Static analysis tools augmented with LLMs can significantly reduce false positive rates in vulnerability reporting, with comparative studies demonstrating improved precision over standalone static analysers [29]. Software quality assurance in alignment with standards such as ISO 25010 is also benefiting from LLM-based techniques. Standards-focused reviews of LLM assurance techniques highlight GPT-4 as consistently outperforming prior baselines across multiple quality dimensions, including reliability, maintainability, and security [28], [44].

6. AI in Software Effort Estimation and Project Planning

Accurate software effort estimation is one of the most persistent and challenging problems in software project management. Traditional models — including COCOMO, Function Points, and Story Points — have been the foundation of project planning for decades. The increasing adoption of LLMs as development assistants challenges these models at a structural level [45], [46].

6.1 LLMs as Effort Estimators

A comprehensive empirical benchmark comparing zero-shot LLM-based estimation with classical ML models and transformer-based regression approaches across story-point and project-level datasets reveals that off-the-shelf LLMs operating in zero-shot settings consistently underperform task-specific ML and transformer-based regression models in terms of absolute accuracy and cost efficiency [17]. Classical ML models remain the most reliable estimators for structured datasets, while fine-tuned transformer regressors achieve the best performance on textual story-point estimation. Despite lower numerical

accuracy, LLMs demonstrate relatively low sensitivity to linguistic variation and may influence human judgment through persuasive explanations, suggesting their utility as qualitative decision-support tools rather than autonomous estimators [17].

6.2 LLM-Aware Effort Estimation Frameworks

The structural limitations of existing estimation models in LLM-mediated development have prompted the conceptualisation of new frameworks. When core development activities are delegated to automated reasoning systems, traditional effort proxies — code size, functional complexity, and task difficulty — become misaligned with actual development cost [18]. Effort increasingly shifts toward interaction management, validation, correction, and integration of LLM outputs. A unified conceptual framework for LLM-aware effort estimation reconceptualises effort as Hybrid Intelligence Effort, emerging from the interaction between LLM cognitive complexity and human oversight effort. Five core dimensions governing effort in LLM-assisted development are identified: LLM reasoning complexity, context and information completeness, code transformation impact, iterative reasoning cycles, and human oversight effort [18]. This framework provides a foundation for developing estimation tools that accurately reflect the economics of modern AI-augmented software development, particularly in agile environments that rely on Story Points.

7. AI in Software Deployment and DevOps

The deployment phase encompasses all activities required to move software from a development environment into production, including continuous integration, continuous delivery, infrastructure provisioning, and monitoring. AI — and increasingly LLMs — are being integrated into DevOps pipelines to automate, optimise, and self-heal these processes [47]–[49].

7.1 Intelligent CI/CD Pipelines

LLM-driven frameworks (LADs) are automating cloud infrastructure management, including the generation of Kubernetes configurations, CI/CD pipeline definitions, and infrastructure-as-code from natural language prompts. What previously required multi-day setup can be accomplished in minutes through conversational interaction with LLM-based DevOps agents [29]. These agents can enforce caching strategies, matrix builds, test stages, and artifact management, and can autonomously refactor configuration files for maintainability.

7.2 Performance Optimisation and Compiler Engineering

At the intersection of AI and system performance, compiler optimisation represents a domain where ML techniques yield measurable gains [50], [51]. Research comparing Intel's OneAPI IFX compiler against Gfortran, NAG, and PGI compilers on Fortran-based systems demonstrates that architecture-specific optimisation can yield improvements of up to 38.4% in execution time without code-level changes [52]. While not AI in the generative sense, such performance research underscores the importance of environment-aware optimisation in deployment — a principle increasingly guided by AI-assisted profiling tools. strategies, matrix builds, test stages, and artifact management, and can autonomously refactor configuration files for maintainability.

8. AI in Software Maintenance

Software maintenance — encompassing corrective, adaptive, perfective, and preventive activities — accounts for the majority of lifetime software costs. AI techniques are being applied to automate several high-cost maintenance tasks including bug localisation, patch generation, technical debt remediation, and documentation update [53], [54].

8.1 Automated Bug Localisation and Repair

LLM-based agents for end-to-end software maintenance follow a common pipeline for resolving real-world issues from repositories such as GitHub. The SWE-bench benchmark evaluates LLM-based agents on their ability to resolve real GitHub issues, capturing the full complexity of production grade software maintenance. Agents such as SWE-agent employ repository navigation, issue reproduction, fault localisation via spectrum-based techniques, and patch generation with iterative validation cycles [29]. The BugStone framework combines LLMs with static program analysis to identify recurring pattern bugs (RPBs) at scale, analysing over 148,000 code artefacts

and processing 117 million input tokens. By summarising patch semantics into coding rules and identifying structurally similar code, BugStone discovers previously unknown instances of known bug patterns — demonstrating that LLMs can augment human expertise in large-scale code quality assurance [29].

8.2 Documentation and Knowledge Management

Keeping software documentation aligned with evolving code is a persistent maintenance challenge. LLMs can automatically generate, update, and verify documentation by comparing docstrings, inline comments, and external documentation against current source code. Systems that generate API documentation from code corpora reduce the manual effort of documentation maintenance while improving accuracy and coverage [55]. Beyond source-code documentation, AI can support the creation and maintenance of requirements specifications, software design documents, user manuals, test reports, release notes, and project knowledge bases [56]. AI systems can also process scanned documents and Optical Character Recognition (OCR) outputs [57], [58], enabling the extraction, summarization, classification, and retrieval of information from legacy documentation archives. By automatically organizing project artifacts, identifying outdated information, and generating concise summaries, AI-driven documentation tools enhance knowledge sharing, improve traceability, and facilitate knowledge retention throughout the software development lifecycle.

9. Challenges and Limitations

9.1 Hallucination and Reliability

Hallucination — the generation of plausible but factually incorrect outputs — is a fundamental limitation of current LLMs. In software engineering contexts, hallucinated code may compile and appear syntactically correct while containing subtle logical errors, security vulnerabilities, or incorrect API usage. Addressing hallucination requires human verification loops, test-driven generation workflows, and domain-specific fine-tuning.

9.2 Context Window and Scalability Constraints

While context windows have expanded from thousands to hundreds of thousands of tokens in frontier models, large codebases can still exceed these limits. Retrieval-augmented generation and hierarchical summarisation strategies partially mitigate this constraint, but fundamental scalability limitations persist for repository-scale tasks.

9.3 Security and Privacy

Transmitting proprietary source code to cloud-hosted LLM APIs raises intellectual property and data privacy concerns. Federated learning frameworks and local model deployment offer privacy-preserving alternatives but involve trade-offs in model capability, infrastructure cost, and maintenance overhead.

9.4 Estimation and Economic Misalignment

As discussed in Section 6, the widespread adoption of LLMs introduces structural misalignment between traditional effort estimation models and actual development costs. Organisations that continue to use COCOMO or Story Points without adaptation risk systematic underestimation or overestimation of project effort in LLM-assisted environments.

9.5 World Model and Reasoning Limitations

Persistent state tracking, causal reasoning, and long-horizon planning remain weak points of current sequence-prediction LLMs. These limitations restrict LLM utility in system design and complex maintenance scenarios, and motivate research into hybrid neuro-symbolic approaches and world models that augment LLM capabilities with structured latent state representations.

CONCLUSION

This review has examined the integration of AI — with particular emphasis on LLMs — across all phases of the SDLC. The evidence is clear: AI is reshaping software engineering in ways that are simultaneously profound and uneven. Code generation and testing have seen the most mature commercial adoption, with tools such as GitHub Copilot demonstrating measurable productivity gains. Requirements engineering is being transformed by LLMs capable of generating SRS documents

comparable to human engineers. Effort estimation frameworks are being reconceptualised to account for the economics of hybrid human-AI development.

At the same time, important limitations constrain the responsible deployment of LLMs in software engineering. Hallucination, context window constraints, privacy concerns, and structural misalignment with established estimation models all demand continued research. The observation that LLMs underperform task-specific models in autonomous effort estimation, while showing potential as qualitative decision-support tools, suggests that a human-centred, hybrid approach is the most pragmatic near-term strategy.

Future research directions include: (i) development of LLM-aware effort estimation frameworks such as Hybrid Intelligence Effort; (ii) design of privacy-preserving fine-tuning pipelines for proprietary codebases using federated learning; (iii) creation of high-quality benchmark datasets — analogous to PlantUCD for RE — for all SDLC phases; (iv) investigation of world models and structured reasoning approaches to address LLM limitations in long-horizon planning; and (v) development of evaluation methodologies that keep pace with rapidly improving LLM capabilities.

In conclusion, the integration of AI into the SDLC is not a distant aspiration but an ongoing industrial and academic reality. Realising its full potential while managing its risks will require interdisciplinary collaboration between software engineering researchers, AI practitioners, and organisational decision-makers.

REFERENCES

1. Bharadiya JP. Artificial intelligence in transportation systems a critical review. American Journal of Computing and Engineering. 2023;6(1):35- 45.
2. Malathi D, Alaswad F, Aljaddouh B, Ranganayagi L, Sangeetha R. AI-Powered Traffic Management: Improving Congestion Detection and Signal Regulation. In: 2025 International Conference on Multi-Agent Systems for Collaborative Intelligence (ICMSCI). IEEE; 2025. p. 899- 904.
3. Christensen J. AI in financial services. In: Demystifying AI for the Enterprise. Productivity Press; 2021. p. 149-92.
4. Alaswad F, Muruganantham B, Bouchi A. Categorising customers and predicting their buying patterns taking into account the customer evaluation model RFM. International Journal of Business Forecasting and Marketing Intelligence. 2021;7(2):124-42.
5. Aljaddouh B, Malathi D, Alaswad F. Multimodal disease detection and classification using breath sounds and vision transformer for improved diagnosis. Procedia Computer Science. 2024;235:1436-44.
6. Shandhi MMH, Dunn JP. AI in medicine: Where are we now and where are we going? Cell Reports Medicine. 2022;3(12).
7. Jayaseeli JD, Malathi D, Alaswad F, Aljaddouh B, Bhardwaj A, Sahay T. Detection of oil-spill in satellite-based synthetic aperture radar (sar) images using neural network. In: 2023 4th IEEE Global Conference for Advancement in Technology (GCAT). IEEE; 2023. p. 1-6.
8. Lazic L. Benefit from Ai in cybersecurity. In: Proc. 11th Int. Conf. Bus. ´ Inf. Secur.(BISEC); 2019. p. 103-19.
9. Grossmann I, Feinberg M, Parker DC, Christakis NA, Tetlock PE, Cunningham WA. AI and the transformation of social science research. Science. 2023;380(6650):1108-9.
10. Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, et al. Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology. 2024;33(8):1- 79.
11. Zhang Q, Fang C, Xie Y, Zhang Y, Yu S, Sun W, et al. A survey on large language models for software engineering. Science China Information Sciences. 2026;69(4):141102.
12. Peng S, Kalliamvakou E, Cihon P, Demirer M. The impact of ai on developer productivity: Evidence from github copilot. arXiv preprint arXiv:230206590. 2023.
13. Hemmat A, Sharbaf M, Kolahdouz-Rahimi S, Lano K, Tehrani SY. Research directions for using LLM in software requirement engineering: A systematic review. Frontiers in Computer Science. 2025;7:1519437.

14. Ferrari A, Spoletini P. Formal requirements engineering and large language models: A two-way roadmap. Information and Software Technology. 2025;181:107697.
15. Alaswad F, Poovammal E. Software quality prediction using machine learning. Materials Today: Proceedings. 2022;62:4714-20.
16. El Haji K, Brandt C, Zaidman A. Using github copilot for test generation in python: An empirical study. In: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024); 2024. p. 45-55.
17. Alaswad F, Poovammal E, Aljaddouh B. From cocomo to gpt: A comprehensive evaluation of llm-based software effort estimation. IEEE Access. 2026.
18. Alaswad F, Poovammal E, Aljaddouh B. Toward LLM-aware software effort estimation: a conceptual framework. Frontiers in Artificial Intelligence. 2026;9:1772418.
19. Krishna M, Gaur B, Verma A, Jalote P. Using llms in software requirements specifications: An empirical evaluation. In: 2024 IEEE 32nd International Requirements Engineering Conference (RE). IEEE; 2024. p. 475-83.
20. Zadenoori MA, Dkabrowski J, Alhoshan W, Zhao L, Ferrari A. Large language models (llms) for requirements engineering (re): A systematic literature review. arXiv preprint arXiv:250911446. 2025.
21. Saleem S, Asim MN, Elst LV, Dengel A. Generative language models potential for requirement engineering applications: insights into current strengths and limitations. Complex & Intelligent Systems. 2025;11(6):278.
22. Rizvi SBA. A SYSTEMATIC MAPPING STUDY ON THE USE OF LLMS FOR REQUIREMENTS TRACEA-BILITY LINK DISCOVERY. Trepo. 2025.
23. Zogaan W, Sharma P, Mirahkorli M, Arnaoudova V. Datasets from fifteen years of automated requirements traceability research: Current state, characteristics, and quality. In: 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE; 2017. p. 110-21.
24. Ibtasham MS. Towards contextually aware large language models for software requirements engineering: A retrieval augmented generation framework. diva-portal. 2024.
25. Alaswad F, Poovammal E, Aljaddouh B, Supriya B. Software Requirements to UML Class Diagrams Using Machine Learning and Rule Based Approach. In: International Conference on Intelligent Systems in Computing and Communication. Springer; 2023. p. 86-101.
26. Alaswad F, E P, Aljaddouh B. PLANTUCD: A DATASET OF SOFTWARE REQUIREMENTS AND CORRESPONDING PLANTUMLBASED CLASS DIAGRAMS. 2026.
27. Ferrari A, Abualhaija S, Arora C. Model generation with LLMs: From requirements to UML sequence diagrams. In: 2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW). IEEE; 2024. p. 291-300.
28. Hong S, Zhuge M, Chen J, Zheng X, Cheng Y, Wang J, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In: International Conference on Learning Representations. vol. 2024; 2024. p. 23247-75.
29. Yang J, Jimenez C, Wettig A, Lieret K, Yao S, Narasimhan K, et al. Swe-agent: Agent-computer interfaces enable automated software engineering. Advances in Neural Information Processing Systems. 2024;37:50528-652.
30. Rasheed Z, Sami MA, Waseem M, Kemell KK, Wang X, Nguyen A, et al. Ai-powered code review with llms: Early results. arXiv preprint arXiv:240418496. 2024.
31. Cheema UI. Effects of artificial intelligence (AI) in software security. lutpub. 2025.
32. McMahan B, Moore E, Ramage D, Hampson S, y Arcas BA. Communication-efficient learning of deep networks from decentralized data. In: Artificial intelligence and statistics. Pmlr; 2017. p. 1273-82.
33. Kairouz P, McMahan HB. Advances and open problems in federated learning. Foundations and trends in machine learning. 2021;14(1-2):1- 210.
34. Jayaseeli JD, Malathi D, Aljaddouh B, Alaswad F, Shah A, Choudhary D. Image classification using federated averaging algorithm. In: 2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS). IEEE; 2023. p. 675-80.
35. Alhumam A, Ahmed S. Software requirement engineering over the federated environment in distributed software development process.

Journal of King Saud University-Computer and Information Sciences. 2024;36(9):102201.

36. Talakola S. The optimization of software testing efficiency and effectiveness using AI techniques. International Journal of Artificial Intelligence, Data Science, and Machine Learning. 2024;5(3):23-34.
37. Banala S, Panyaram S, Selvakumar P. Artificial intelligence in software testing. In: Artificial intelligence for cloud-native software engineering. IGI Global Scientific Publishing; 2025. p. 237-62.
38. Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering. 2024;50(4):911-36.
39. Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, et al. Large language models for software engineering: Survey and open problems. In: 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). IEEE; 2023. p. 31-53.
40. Oveisi S, Moeini A, Mirzaei S, Farsi MA. Software reliability prediction: A survey. Quality and Reliability Engineering International. 2023;39(1):412-53.
41. Jha S, Kumar R, Abdel-Basset M, Priyadarshini I, Sharma R, Long HV, et al. Deep learning approach for software maintainability metrics prediction. Ieee Access. 2019;7:61840-55.
42. Lu G, Ju X, Chen X, Pei W, Cai Z. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. Journal of Systems and Software. 2024;212:112031.
43. Taghavi Far SM, Feyzi F. Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics. International Journal of Information Security. 2025;24(2):78.
44. Patil A. Advancing Software Quality: A Standards-Focused Review of LLM-Based Assurance Techniques. arXiv preprint arXiv:250513766. 2025.
45. Bui TL, Dam HK, Hoda R. An LLM-based multi-agent framework for agile effort estimation. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2025. p. 1032-43.
46. Yonathan M. Explainable Local LLMs for Agile Sprint Effort Estimation: A Reproducible Proof of Concept with Benchmarks. Available at SSRN 5639171. 2025.
47. Pahl C, Sezen OC, Hofer F. Artificial Intelligence for Infrastructure-as- ¨ Code—A Systematic Literature Review. Electronics. 2026;15(4):755.
48. Adebayo O. From Test Automation to Intelligent QA: Leveraging AI in Quality Assurance for RPA and DevOps. International Journal of Computer Techniques.
49. Khan J. Leveraging Artificial Intelligence to Automate ETL Pipelines: Evolving Legacy Data Systems into Intelligent Workflows. researchgate. 2025.
50. Bian K, Priyadarshi R. Machine learning optimization techniques: a survey, classification, challenges, and future research issues. Archives of Computational Methods in Engineering. 2024;31(7):4209-33.
51. Azevedo BF, Rocha AMA, Pereira AI. Hybrid approaches to optimization and machine learning methods: a systematic literature review. Machine Learning. 2024;113(7):4055-97.
52. Alaswad F, Eswaran P. Investigating the superiority of Intel oneAPI IFX compiler on Intel CPUs using different optimization levels: A case study on a CFD system. In: 2023 4th IEEE Global Conference for Advancement in Technology (GCAT). IEEE; 2023. p. 1-9.
53. Puvvadi M, Arava SK, Santoria A, Chennupati SSP, Puvvadi HV. Coding agents: A comprehensive survey of automated bug fixing systems and benchmarks. In: 2025 IEEE 14th International Conference on Communication Systems and Network Technologies (CSNT). IEEE; 2025. p. 680-6.
54. Ajeigbe K, Emma O. Dynamic Documentation Generation with AI. researchgate. 2024.
55. Alrefai A, Alsadi M. Large Language Models for Documentation: A Study on the Effects on Developer Productivity; 2024.
56. Burte S. AI-Powered Software Development Life Cycle: From Requirements to Maintenance. AI Systems Engineering. 2025;1(1):1-8.

57. Aydın O. Classification of documents extracted from images with optical ¨ character recognition methods. Computer Science. 2021;6(2):46-55.
58. Alaswad F, Poovammal E, Issa H. Off-Line Recognition System for Handwritten Arabic Words Using Artificial Intelligence. In: 2021 8th International Conference on Signal Processing and Integrated Networks (SPIN). IEEE; 2021. p. 556-61.

HOW TO CITE: A. Mohanlal, Feisal Alaswad*, Batoul Aljaddouh, The Generative SDLC: A Systematic Review of Integrating Modern LLMs in Software Development Life-Cycle, Int. J. Sci. R. Tech., 2026, 3 (6), 911-921. https://doi.org/10.5281/zenodo.20701463