Towards the Integration of Large Language Models into the Software Development Life Cycle: A Systematic Literature Review

1st Mohammadamin Madani
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany
mohammadamin.madani@st.ovgu.de

2nd Ksenia Neumann
Very Large Business Application Lab
Otto von Guericke University
Magdeburg, Germany
ksenia.neumann@ovgu.de

3rd Abdulrahman Nahhas
Very Large Business Application Lab
Otto von Guericke University
Magdeburg, Germany
abdulrahman.nahhas@ovgu.de

4th Maria Chernigovskaya
Very Large Business Application Lab
Otto von Guericke University
Magdeburg, Germany
maria.chernigovskaya@ovgu.de

5th Damanpreet Singh Walia
Very Large Business Application Lab
Otto von Guericke University
Magdeburg, Germany
damanpreet.walia@ovgu.de

6th Klaus Turowski
Very Large Business Application Lab
Otto von Guericke University
Magdeburg, Germany
klaus.turowski@ovgu.de

***Abstract*—Large Language Models (LLMs) are reshaping the Software Development Life Cycle (SDLC) by introducing new levels of automation and intelligent assistance across core phases, including requirements engineering, design, testing, and maintenance. Previous works have examined the adoption of LLM in these areas, but have overlooked its influence on traditional practices in SDLC. Therefore, this paper builds upon this foundation and investigates, from a phase-level perspective, how LLMs interact with the SDLC as a whole and whether their presence necessitates rethinking of conventional methodologies. To achieve this objective, this article reviews the state of the field using a systematic literature review method. Peer-reviewed studies published between 2019 and 2025, examining the integration, benefits, and limitations of LLM within SDLC are analyzed. The findings indicate that LLMs are most commonly applied in implementation and testing, where they improve productivity through code generation, test automation, and fault localisation. However, their adoption also introduces challenges related to model hallucination, contextual misalignment, computational overhead, and ethical risks such as bias and opacity. Therefore, we synthesise several LLM integration patterns and stress various risk mitigation strategies that can be adopted to enhance the integration of LLMs into software engineering practices. The study concludes by outlining practical implications for software teams and proposing future research directions to improve LLM's integration with agile, DevOps and CI/CD workflows.**

***Index Terms*—Large Language Models, Software Development Life Cycle, Systematic Literature Review, Generative AI, Software Engineering, SDLC, Software Automation, LLM Challenges**

I. Introduction

The emergence of Large Language Models (LLMs), such as GPT-3 and GPT-4, is reshaping software engineering by influencing how software is planned, developed, tested, and maintained [1], [2]. Their ability to understand and generate software-related artefacts enables new forms of automation, accelerates delivery, and challenges established workflows [3]. As these models increasingly permeate the Software Development Life Cycle (SDLC), it becomes essential to examine both the opportunities they unlock and the risks they introduce. The SDLC comprises structured phases —requirements gathering, design, development, testing, deployment, and maintenance — that have historically relied on manual practices or narrowly-scoped automation. With LLMs now supporting everything from requirements clarification to test generation, the nature of work across phases is changing [4]. In practice, "integration" spans a spectrum: at one end, LLMs assist with discrete developer tasks (e.g., code scaffolding, refactoring, drafting tests); in the middle, they augment specific phases (e.g., ambiguity detection in requirements, deriving design artefacts, composing CI/CD scripts); and at the other end, they help orchestrate artefacts and feedback across phases (e.g., tracing requirements through generated acceptance tests and feeding test outcomes back into validation). Alongside efficiency gains, concerns about reliability, bias, explainability, and governance remain central [5]. Prior work has explored LLM-enabled code generation, testing, and security [1], [6]. Yet less attention has been paid to how phase-level practices and cross-phase dependencies are reshaped, or to recurring ways organisations embed LLMs into day-to-day engineering. To address this, we adopt a phase-aware perspective on SDLC and synthesise recurrent patterns of use, while also considering how these practices align with Agile, DevOps, and CI/CD.

A. Related Work

Recent Systematic Literature Reviews (SLRs) have explored LLMs in Software Engineering (SE), providing a foundation for our study. Hou et al. [7] conducted an SLR on 395 studies (2017–2024), categorizing LLMs by architecture, pretraining, and SE tasks like code generation, while highlighting optimisation strategies and evaluation gaps. Zhang et al. [1] surveyed 62 LLMs and 947 studies across code-related tasks in five SE phases, discussing empirical evaluations, security, and future directions like domain tuning. Fan et al. [4] provided a survey on LLMs in SE, covering applications from coding to analytics, and identifying open challenges like hallucination mitigation through hybrid techniques. While these SLRs share our use of databases (e.g., IEEE Xplore) and PRISMA protocols, they primarily adopt broad, task-oriented lenses, often overlooking SDLC's structured phases and cross-phase dependencies. In contrast, our phase-aware analysis synthesises integration patterns (e.g., from assistant tools to CI/CD orchestration) and mitigation strategies, addressing gaps in traditional methodology adjustments and workflow alignment (e.g., Agile/DevOps). This positions our work as a novel contribution, guiding LLM adoption across SDLC phases. This review is guided by the following research questions:

- **RQ1** How do different phases of SDLC change with the integration of LLMs, and to what extent might the traditional SDLC model need to be adjusted?
- **RQ2** What technical, ethical, and operational challenges arise when incorporating LLMs into the SDLC?
- **RQ3** How can software development processes be continuously adapted to align with the evolving role of LLMs in SDLC?

RQ1 examines per-phase changes and the sufficiency of existing methodologies; RQ2 investigates risks that include precision, security and influence on decision making [8], [9]; and RQ3 focuses on sustainable adoption through adaptable engineering practices and governance strategies. Together, these questions link progress in generative AI with practical software engineering, synthesise current evidence, highlight underexplored areas, and offer guidance for integrating LLMs across SDLC.

II. Methodology

This study employs an SLR to investigate how LLMs impact activities and artefacts across the SDLC. The protocol—encompassing question formulation, search and screening, eligibility assessment, and synthesis—was designed for replicability and transparency, and is documented via a PRISMA flow diagram (Figure 1) [10].

A. Search Strategy and Database Selection

The literature search was conducted between October 2024 and January 2025, targeting peer-reviewed studies on LLM integration within SDLC phases. We selected four key digital libraries that index core software engineering and AI venues:

- **IEEE Xplore** (engineering, computing, and AI applications),
- **ACM Digital Library** (software engineering and programming research),
- **SpringerLink** (theoretical and applied computer science),
- **ScienceDirect** (peer-reviewed AI and SE studies).

Broader indexers (e.g., Scopus, Web of Science) were considered during scoping; however, observed overlap with the selected sources, combined with access and deduplication constraints, led us to prioritize these four. Potential coverage gaps are acknowledged in the limitations section.

To optimize recall and precision, we iteratively refined search terms through pilot runs and synonym expansion, guided by best practices [11]. Queries were applied to titles, abstracts, and keywords using Boolean operators and database-specific filters. The primary query focused on phase-aware SDLC terms alongside LLM-related nomenclature, while two complementary strings expanded coverage to productivity/automation and risk/ethics. The final search strings are summarized in Table I.

TABLE I: Search strings (final) used in the study.

Search String	Content
Main	(”Impact” OR ”Influence” OR ”Effect”) AND (”LLM” OR ”Generative AI” OR ”GenAI” OR ”Artificial Intelligence”) AND (”SDLC” OR ”Software Development Process” OR ”Systems Development Life Cycle” OR ”Requirements Analysis” OR ”Design Phase” OR ”Testing Phase” OR ”Deployment Phase”)
S2	(”Software Engineering” OR ”Software Development”) AND (”AI” OR ”Artificial Intelligence” OR ”Large Language Model” OR ”Transformer Model”) AND (”Automation” OR ”Productivity” OR ”Workflow”)
S3	(”Generative Model” OR ”LLM” OR ”Large Language Model”) AND (”Challenge” OR ”Risk” OR ”Ethical Issue”) AND (”Software Life Cycle” OR ”SDLC” OR ”Development”)

B. Search Parameters and Study Selection

Eligible records were restricted to peer-reviewed journal and conference papers published between 2019 and 2025. Searches targeted titles, abstracts, and keywords, with the language limited to English. Inclusion and exclusion criteria are detailed in Table II. The selection process involved multiple stages: automated deduplication, title/abstract screening, criteria-based filtering, and full-text evaluation for eligibility and phase-specific relevance. Disagreements were resolved through discussion following a calibration pass on a subset of records.

To ensure the reliability and validity of the search and selection processes, we adopted structured measures. Search terms underwent pilot testing on a sample of 50 records from IEEE Xplore, initially yielding a recall of 82% and precision of 78%, which were refined through synonym expansion and team feedback to achieve a balanced 85% recall and 80% precision. The selection process's validity was evaluated by two independent reviewers who assessed 30 abstracts, achieving an inter-rater agreement of 92% (Cohen's Kappa = 0.88), with discrepancies resolved through consensus. Furthermore, the quality appraisal phase employed a checklist of five criteria — clarity of reporting, methodological rigor, relevance to RQ1-3, evidence strength, and bias disclosure — applied to all 76 full-text studies to maintain a consistent inclusion threshold.

These steps enhance the reproducibility and robustness of our SLR methodology.

TABLE II: Inclusion and exclusion criteria.

Inclusion	Exclusion
LLM studies mapped to SDLC phases	Work not situated in SDLC
Peer-reviewed journals/conferences	Non-peer-reviewed sources
2019–2025 (plus seminal if relevant)	General AI without SDLC grounding
Quantitative, empirical, or theoretical analyses; reviews with SDLC lens	Domain-external applications without SE linkage

C. Screening and Selection Overview

The selection process is summarized in the PRISMA flow diagram (Figure 1). Key steps include:

- **Initial retrieval:** 1,007 records (535 main; 472 complementary).
- **Deduplication:** 662 duplicates removed; 345 unique studies remained.
- **Title/abstract screening:** 123 excluded.
- **Criteria filtering:** 146 excluded per Table II.
- **Full-text assessment:** 76 reviewed; 44 excluded; 32 retained.

D. Quality Appraisal and Data Extraction

To ensure consistent synthesis, retained studies underwent a lightweight quality appraisal based on clarity of reporting, methodological soundness, and relevance to the research questions. Studies failing to meet a preset threshold during full-text review were excluded. Data extraction captured: (i) SDLC phase(s) addressed, (ii) task and artefact focus, (iii) study type and evidence (e.g., experiment, case, benchmark), (iv) reported outcomes (productivity/quality) and risks (e.g., hallucination, bias, cost), and (v) integration mechanisms (e.g., assistant tools, pipeline steps, CI/CD hooks). The research questions were framed using a PICOC lens [11]: *Population* (peer-reviewed SE studies), *Intervention* (LLM integration within SDLC activities), *Comparison* (conventional practices, where applicable), *Outcome* (impacts and challenges), and *Context* (SDLC phases), ensuring a phase-aware and outcome-oriented approach.

III. Literature Analysis

This section reports the SLR findings over the final set of 32 peer-reviewed studies. We organise the evidence by SDLC phase and then synthesise recurring cross-phase challenges. In contrast to broad tool-centric surveys, our synthesis is explicitly phase-aware and attentive to recurring patterns of use across organisational workflows.

A. Findings by SDLC Phase

The selected studies span multiple SDLC phases, with *testing* and *implementation* most frequently addressed (Figure 2).

a) Requirements: LLMs assist with extracting and clarifying requirements, detecting ambiguities, and converting natural-language specifications into structured artefacts [12], [13]. They have been explored for formal analysis and modelling of requirements as well, improving early detection of inconsistencies and incompleteness [9]. Recent work also examines direct transitions from requirements to code under controlled settings, streamlining handover into downstream phases [13]. Despite these advances, adaptation to specialised domains remains challenging and often calls for careful prompting or hybrid strategies [8].

b) Design: Evidence in the design phase is comparatively limited. Studies explore deriving design artefacts and planning transformations from textual prompts or higher-level intent [14], [15]. Open challenges include scaling abstraction, ensuring usability in collaborative settings, and validating design decisions under realistic constraints [16], [17].

c) Implementation: LLMs support code generation, refactoring, and boilerplate creation, and are increasingly embedded into developer workflows [18], [19]. Program improvement and repair pipelines (e.g., autonomous refinement loops) show promise but remain sensitive to prompt and context management [6], [19]. Hallucinated or semantically inconsistent code persists as a risk, motivating stronger guardrails and evaluation protocols [4], [20].

d) Testing: Testing is the most extensively studied phase. Reported uses include unit-test improvement and generation, fault localisation, and program repair [6], [21], [22]. Prompting strategies (e.g., few-shot) can increase relevance and coverage, though reliability on edge cases and complex integrations remains a concern [23], [24]. Empirical results indicate benefits, but also highlight the need for a rigorous and repeatable evaluation [22], [25].

e) Deployment: Across delivery pipelines, LLMs are investigated for CI/CD support, such as composing configuration scripts, policy-as-code, and automating routine release tasks, while introducing new concerns about verification and rollback safety [5]. Lifecycle-oriented discussions in LLMOps further emphasise governance, observability, and risk controls during and after deployment [3], [26].

f) Maintenance: In maintenance contexts, LLMs aid bug localisation and fix suggestion, vulnerability repair, and incremental code improvement [6], [19], [27]. They are also used to streamline recurring changes (e.g., systematic refactorings) and to assist with legacy remediation. While case studies report promising outcomes, large-scale validations on complex codebases remain sparse.

B. Cross-Phase Challenges

Recurring themes emerge across phases, revealing broader limitations and unresolved concerns in integrating LLMs into software engineering workflows:

- **Ethical and operational risks.** Outputs can exhibit bias, hallucination, or misinterpretation and often lack transparency/auditability, underscoring the need for governance and explainability [16], [17]. In privacy-sensitive

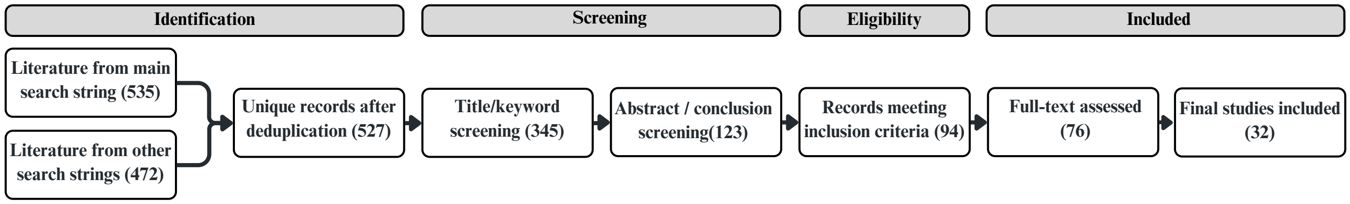

Fig. 1: PRISMA flow diagram for the study selection process.

settings, assured or offline deployments are increasingly considered [28].

- **Scalability and domain adaptation.** General-purpose models struggle in niche or regulated domains; domain-aware prompting, fine-tuning, and hybrid pipelines are common mitigations [1].
- **Empirical gaps.** Many studies rely on controlled settings, limiting external validity. Broader industrial evaluations and standardised benchmarks remain a need [7], [29].
- **Research imbalance across phases.** The literature concentrates on implementation/testing, while design, deployment, and maintenance are under-explored and warrant deeper, phase-balanced studies [1], [22].
- **Human–AI collaboration.** Human-in-the-loop workflows mitigate automation errors and support contextual decision-making, improving practical utility [8], [18].
- **Productivity vs. skill development.** Efficiency gains in routine tasks can be offset with long-term skill formation; balanced workflows are advised [30].

These cross-phase challenges capture a dual trajectory: tangible efficiencies alongside open questions on scalability, ethical robustness, and empirical maturity. The next section consolidates the phase-wise contributions and outlines research opportunities.

IV. Contributions of LLMs Across SDLC Phases

This section synthesises the most salient contributions of LLMs across the SDLC. Large Language Models automate a range of manual tasks, accelerate delivery cycles, and augment human creativity— thereby reshaping foundational software engineering practices [1], [8].

A. Distribution of Research Across SDLC Phases

Our review of 32 peer-reviewed studies indicates that LLM integration is most extensively explored in **testing and debugging** (25%), followed by **implementation** (21.8%) and **requirements engineering** (18.4%). In contrast, phases such as **design**, **deployment**, **maintenance**, and **operations/service management** are comparatively less represented.

Figure 2 illustrates the phase-wise distribution of research efforts, highlighting a concentration on mid-phase engineering tasks where code generation, fault localisation, and test automation dominate current applications of LLMs. This imbalance suggests opportunities for deeper investigation into under-explored phases such as early design ideation, post-deployment monitoring, and runtime service adaptation.

B. Concept Matrix of LLM Contributions

Table III outlines typical roles, benefits, and limitations of LLMs across SDLC phases, aggregating insights from the analysed studies and marking areas that warrant further research.

TABLE III: Concept Matrix: Contributions of LLMs Across SDLC Phases

SDLC Phase	Key Contributions	Studies
Requirements Engineering	Automates requirement analysis, ambiguity detection, validation, and transformation of natural language into structured artefacts.	[4], [7], [12], [13]
Design	Derives design artefacts (e.g., UML), pattern suggestions, and planning from textual intent; open issues include scalability and usability.	[14]–[16]
Implementation	Code generation, refactoring, boilerplate scaffolding, and collaborative coding support within developer workflows.	[18]–[20], [24]
Testing and Debugging	Test generation/improvement, fault localisation, and program repair; benefits tempered by edge-case reliability concerns.	[6], [21]–[23]
Deployment	CI/CD support (pipeline/config generation), policy-as-code, and operational safety checks during delivery.	[3], [5], [15]
Maintenance	Bug localisation/fixing, vulnerability repair, and incremental code improvement; large-scale industrial evidence remains limited.	[6], [19], [27]
Operations/Service Management (post-deployment)	Assists with documentation, incident reports, and orchestration/diagnostics in service-oriented settings.	[20], [28], [29]

C. Discussion of Findings by SDLC Phase

The integration of LLMs varies by phase. Earlier phases benefit from transforming natural language into structured artefacts; later phases leverage generative capabilities for automation, testing, and operational support. Figure 3 summarises representative contributions — such as ambiguity detection in requirements, architecture suggestions in design, code scaffolding in implementation, test case generation in testing, CI/CD automation in deployment, and diagnostics in operations.

- **Requirements Engineering:** Extraction/clarification of requirements, ambiguity detection, and transformation

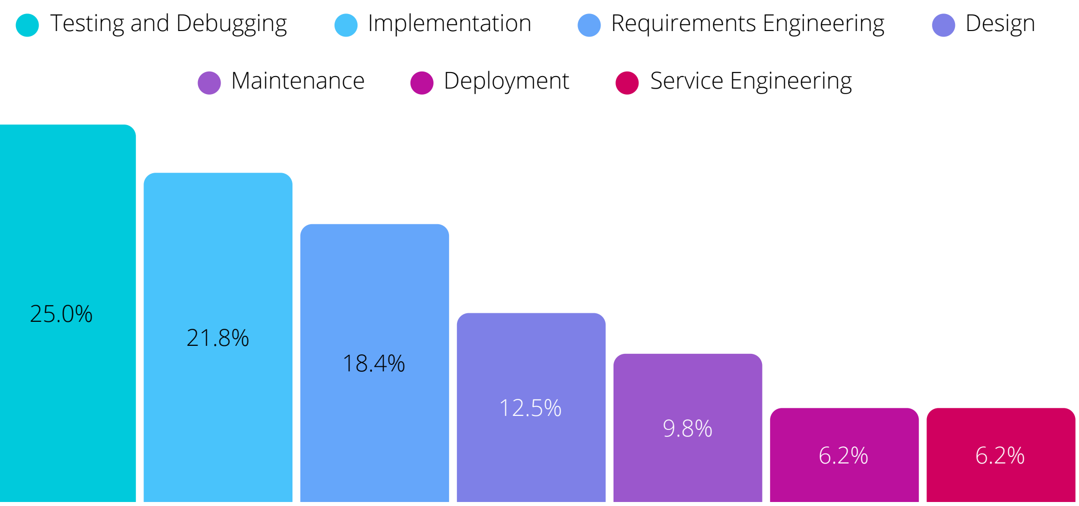

Fig. 2: Distribution of studies across SDLC phases. Testing and implementation dominate, with early (requirements, design) and late (deployment, maintenance) phases receiving comparatively less attention.

into actionable items; selective use for formal artefacts and requirements-to-code handover under controlled conditions [12], [13].

- **Design:** Deriving design artefacts and pattern recommendations from textual intent; challenges include scaling abstraction and validating design decisions collaboratively [14]–[16].
- **Implementation:** Code generation and refactoring, boilerplate scaffolding, and documentation support; risks include hallucinated or semantically inconsistent snippets, motivating stronger guardrails and evaluation [18]–[20].
- **Testing and Debugging:** Unit-test generation/improvement, mutation testing support, fault localisation, and repair; prompting strategies can improve relevance, but edge-case reliability remains an issue [6], [21]–[23].
- **Deployment:** Assistance with CI/CD configuration, infrastructure/policy-as-code authoring, and routine release tasks; governance and rollback safety remain active concerns [3], [5].
- **Maintenance:** Support for bug localisation/fixes and incremental improvements, including legacy remediation; broader industrial validations are still sparse [6], [27].
- **Operations/Service Management:** Documentation and incident handling, ChatOps-style diagnostics, and orchestration guidance in complex service landscapes [20], [28].

D. *Effectiveness and Limitations*

1) *Strengths:*

- **Automation:** Reduced manual effort in repetitive development/testing tasks [21].
- **Collaboration:** IDE-integrated assistants and code copilots can support team workflows and knowledge sharing [18].
- **Time efficiency:** Faster prototyping and delivery, with reported productivity gains in case studies [30].
- **Distributed practice:** Documentation, review, and testing support teams across locations and time zones [31].

2) *Challenges:*

- **Contextual accuracy:** Models may misinterpret nuanced or domain-specific inputs [22].
- **Ethical and safety risks:** Bias, hallucination, and overreliance highlight the need for governance and explainability [16], [17].
- **Computational overhead:** Resource demands can hinder adoption in constrained environments [5].
- **Legacy integration:** Compatibility with older systems requires tailored adaptation strategies [20].

V. Practical Insights

Although much of the academic literature evaluates the capabilities of LLMs within the SDLC, practical adoption in real-world settings introduces dimensions that extend beyond algorithmic performance. This section synthesises applied knowledge from empirical studies and case reports to inform practitioners about actionable patterns, tooling options, and integration strategies.

Our analysis highlights three layers of practical impact: (1) tooling integration and selection, (2) workflow transformation and collaboration, and (3) deployment governance and oversight. Each layer offers immediate productivity benefits but also entails longer-term architectural and organisational changes that practitioners should anticipate.

A. *Tooling Landscape*

Prominent LLM-enabled tools used in practice can be grouped into developer-facing assistants and infrastructure-level platforms.

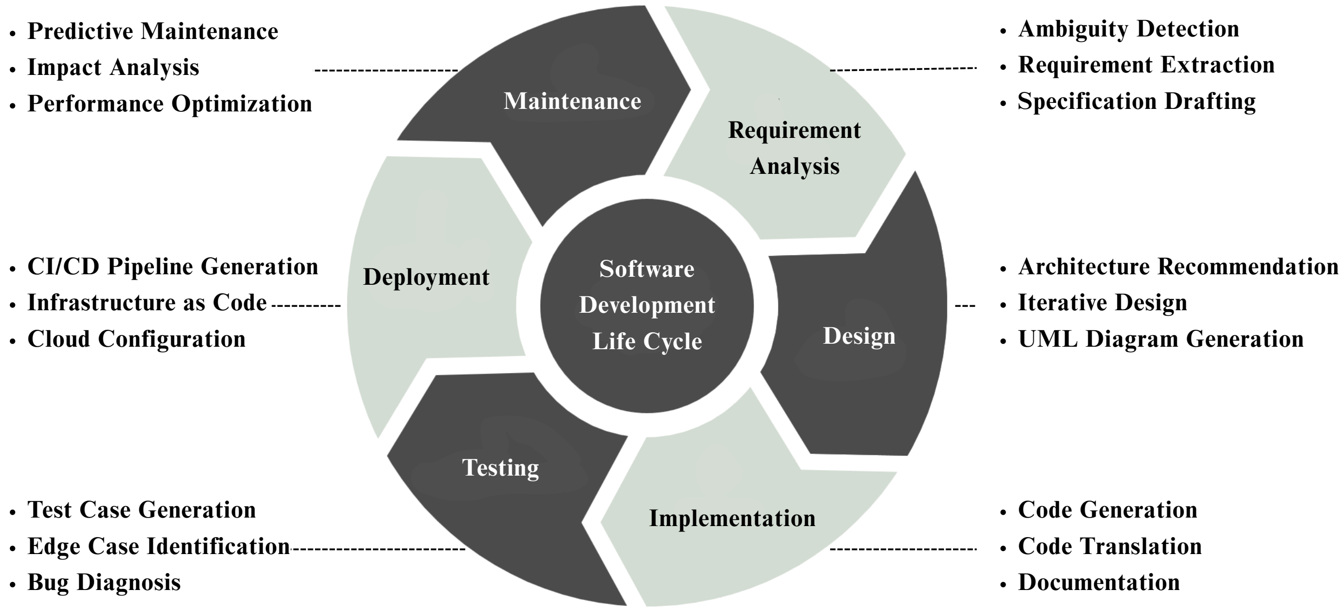

Fig. 3: Key contributions of LLMs across SDLC phases.

1) Developer-Facing Assistants: IDE-integrated assistants provide autocomplete, comment-to-code, refactoring suggestions, and test scaffolding. Studies report time savings on boilerplate and unit-test authoring, alongside improvements in routine coding tasks [18], [21]. However, reliance on suggestions for novel algorithms risks admitting logically incorrect or hallucinated code; therefore, teams typically treat suggestions as intelligent scaffolding and require review before merge [4], [20].

2) Infrastructure and DevOps Tooling: LLMOps practices and platforms support fine-tuning, evaluation, deployment, and monitoring of models, including artefact/version management for prompts and policies [3], [26]. In CI/CD contexts, these controls underpin rollback safety and policy-as-code checks when LLM steps are part of the delivery pipelines [5]. Emerging practices such as prompt chaining, role-specific agent workflows, and embedding-aware feedback loops are increasingly reported in QA and post-deployment analysis [3], [7].

B. Integration Patterns Across the SDLC

We observe four recurring integration patterns that characterise how LLMs are embedded in today's workflows:

1) Pattern 1: Task-Specific Assistance: LLMs assist discrete, well-scoped tasks — e.g., summarising requirement documents, generating unit tests, or proposing refactorings [12], [21]. Integration is usually manual or semi-automated and does not mandate structural changes. *Example:* a developer invokes an IDE-integrated assistant to draft tests, then edits before commit.

2) Pattern 2: Phase-Oriented Augmentation: Models are integrated into a specific SDLC phase. During design, for instance, they produce architectural sketches or documentation from prompts; in testing, they generate regression tests on pull requests [16]. This pattern requires prompt templates and validation rules but remains modular and reversible.

3) Pattern 3: Cross-Phase Orchestration: LLMs bridge phases by transferring context — e.g., deriving acceptance tests from requirements and tracing outcomes back to validation. This demands data-flow management, versioning, and feedback loops [22]. Maintaining context consistency and traceability remains challenging.

4) Pattern 4: LLM-Centric Development Pipelines: In AI-native settings, LLMs sit at the centre of the lifecycle (requirements interpretation, scaffolding, documentation, testing, deployment scripting), governed by human-in-the-loop gates and audit trails [31]. The approach maximises automation but amplifies risks related to explainability, hallucination, and drift.

C. Risk Management and Best Practices

1) Risk 1: Hallucinations and Inaccuracies: Syntactically valid yet semantically incorrect outputs can introduce faults or vulnerabilities [5]. **Mitigation:** multi-step validation (static/dynamic analysis), differential checks against ground truth, and mandatory human review for production-critical changes [22].

2) Risk 2: Over-Reliance and Loss of Expertise: Excessive dependence may attenuate critical judgement, particularly for junior developers [23]. **Mitigation:** strong review culture, mentoring, and exercises that require deeper reasoning.

3) Risk 3: Ethical and Regulatory Concerns: Bias, opacity, and potential IP issues necessitate accountability and auditability [16], [17]. **Mitigation:** explainability practices, audit logs of LLM-assisted decisions, and preference for licensed or in-house models where appropriate.

4) Risk 4: Operational and Integration Overhead: Integrating LLMs adds infrastructure complexity (GPU capacity, access control, prompt/artefact registries) [3], [5]. **Mitigation:** containerised deployments, inference as managed services, and LLMOps controls for promotion/rollback.

5) Risk 5: Contextual Drift and Prompt Sensitivity: Small prompt changes can alter results, affecting reproducibility. **Mitigation:** standardised prompt templates, versioned prompt/output pairs, and review processes; treat prompts as first-class artefacts [4], [7].

D. Implications for Organisations and Teams

1) Organisational Culture: Adoption benefits from transparency, cross-functional collaboration, and continuous upskilling [16]. **Recommendation:** AI literacy (prompting, evaluation, ethics) across roles; joint teams spanning engineering, data, and legal.

2) Evolving Developer Roles: As routine tasks are automated, roles shift towards validation, system-level reasoning, and prompt optimisation. **Recommendation:** reflect these expectations in role definitions, hiring, and performance criteria.

3) Team-Level Process Changes: LLM use affects agile ceremonies, VCS practices, and reviews. **Recommendation:** explicitly label AI-originated changes and assess their impact in reviews/retrospectives [4].

4) Tooling and Governance: LLM usage creates the need for governance layers monitoring prompt quality, usage patterns, and integration behaviour. **Recommendation:** internal policies for prompt registries, audit logs, fallback protocols, and LLM-aware plugins in DevOps toolchains [3], [26].

5) Ethical Alignment and Public Perception: Responsible practices influence reputation and compliance [8]. **Recommendation:** document AI policies, publish fairness audits, and provide clear escalation paths for problematic outputs.

E. Alignment with Research Questions and Final Reflections

RQ1. Requirements benefit from ambiguity detection and transformation to structured artefacts; design gains prototyping support; implementation, testing, and debugging see the highest automation; deployment/operations are emerging via CI/CD hooks and observability. These shifts suggest a more fluid, AI-assisted lifecycle.

RQ2. Challenges span technical (domain specificity, long-horizon consistency), operational (costs, governance, process change), and ethical (bias, explainability, over-reliance) dimensions — demanding both technical and organisational responses.

RQ3. Sustainable adoption involves embedding LLM steps in CI/CD and testing environments, treating prompts as versioned artefacts, and instituting LLMOps controls with human-in-the-loop gates [5], [26]. Organisations and AI capabilities should co-evolve to remain aligned.

Final Reflections: LLM integration is transformative rather than superficial. Its spread across phases reshapes implementation practice and team coordination alike. Progress will hinge on pairing automation with accountability; the insights above offer a practical footing for responsible, phase-aware adoption in modern software engineering.

VI. Threats to Validity

Selection and coverage. We queried four domain-salient libraries (IEEE Xplore, ACM DL, SpringerLink, ScienceDirect). Although broader indexers (e.g., Scopus, Web of Science) often increase breadth and deduplication quality, observed overlap and access constraints led us to prioritise these four. Residual coverage gaps are possible.

Publication bias. Industrial deployments and negative results are underrepresented in peer-reviewed venues, which may skew evidence towards positive or research-convenient settings.

Search-string sensitivity. We mitigated sensitivity via pilot refinements and synonym expansion [11]; nonetheless, alternative formulations might surface additional studies.

Screening/researcher bias. Despite predefined criteria, a calibration pass, and a lightweight quality appraisal [11], selection and coding involve judgement; some subjectivity is unavoidable.

Temporal validity. The window (2019–2025) and search period (Oct 2024–Jan 2025) bound our conclusions. Given rapid LLM evolution, newer results may shift the phase-level balance and practices.

VII. Conclusion

This SLR examined how LLMs are being integrated across the SDLC and what impacts, risks, and opportunities emerge. Across the final set of 32 peer-reviewed studies, we find that LLMs automate recurring tasks, accelerate delivery, and augment developer decision-making, while also introducing governance, reliability, and integration challenges that require phase-aware controls.

A. Summary of Key Findings

LLMs show tangible contributions in requirements, design, implementation, testing, and maintenance, with *testing and debugging* and *implementation* currently the most mature areas. In requirements engineering, models assist with ambiguity detection and the transformation of natural language into structured artefacts [12], [13]. In design, early evidence points to generating sketches and design artefacts from textual intent, though scalability and collaborative validation remain open issues [14], [16]. During implementation, LLM-based assistants support code scaffolding, documentation, and refactoring, but remain sensitive to prompt/context management [18]. For testing and debugging, studies report improvements in test generation, fault localisation, and program repair [6], [21]. Post-deployment, emerging work targets CI/CD hooks and operational safeguards to manage model-driven changes [5].

Key takeaways.

- Evidence is densest in *implementation* and *testing*; *design*, *deployment*, and *maintenance* remain comparatively under-explored.

Strategic Enhancement	Description	Key References
Customisation and Fine-Tuning	Tailored datasets and iterative refinement increase contextual accuracy and reliability of outputs.	[5], [32]
Lightweight Architectures	Pruned/quantised models improve scalability and accessibility for smaller teams.	[33]
Ethical Governance	Policies, bias checks, and fairness audits build trust and accountability.	[17]
Training and Development	Upskilling in prompting, evaluation, and responsible use supports effective integration.	[31]

TABLE IV: Strategic enhancements for integrating LLMs into the SDLC.

- Salient risks include *hallucination*, *context drift/bias*, and *operational overhead*, motivating human-in-the-loop checks, prompt/version control, and LLMOps practices [5], [16].
- Integration patterns range from task-specific assistance to cross-phase orchestration; alignment with *Agile/DevOps/CI/CD* helps operationalise these patterns responsibly.

B. Implications of Findings

The findings signal a shift from sequential, manual processes towards more continuous, AI-assisted workflows. Productivity gains are clearest in routine coding and testing tasks; however, expert oversight remains essential—LLMs function best as *co-pilots*, not replacements. Effective adoption therefore hinges on governance (traceability, auditability), evaluation (repeatable checks for reliability and security), and platform practices that manage prompts, artefacts, and rollbacks in delivery pipelines [5], [26].

C. Limitations and Scope

This review synthesises 32 peer-reviewed studies and does not capture all industrial or grey-literature deployments. The search window (October 2024–January 2025) bounds recency. The evidence base is also skewed towards widely available model families of the period, potentially underrepresenting alternatives. Methodological threats (coverage, publication bias, query sensitivity, researcher judgement, temporal validity) are discussed in the dedicated *Threats to Validity* section.

D. Future Research Directions

We outline directions that would strengthen phase-aware, responsible adoption:

- **Phase-balanced evaluation and benchmarks:** Establish standard tasks/datasets for under-explored phases (design, deployment, maintenance) with reproducible pipelines.
- **Context grounding and adaptation:** Improve domain specificity via retrieval-augmented generation and fine-tuning, with transparent prompt/version governance.
- **Operational robustness:** Integrate LLM steps into CI/CD with policy-as-code, rollback safety, and observability; advance lightweight/efficient models for constrained settings.
- **Human–AI collaboration:** Design HITL workflows and metrics that reward review quality, not only speed; study long-horizon effects on maintainability and team skills.
- **Modernisation and migration:** Investigate LLM-supported codebase migration and systematic refactoring at scale as a bridge between legacy and modern stacks.

E. Concluding Remarks

LLMs are becoming integral to how software is conceived, constructed, and operated. Realising their benefits safely requires coupling automation with accountability — combining phase-aware evaluation, governance, and team practices. This review provides a structured map of current evidence and gaps to guide both practitioners and researchers towards responsible, impact-focused integration of LLMs across the SDLC.

REFERENCES

[1] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, "A survey on large language models for software engineering." [Online]. Available: http://arxiv.org/pdf/2312.15223v2

[2] P. Khlaisamniang, P. Khomduean, K. Saetan, and S. Wonglapsuwan, "Generative ai for self-healing systems," in *2023 18th International Joint Symposium on Artificial Intelligence and Natural Language Processing (iSAI-NLP)*. IEEE, 2023, pp. 1–6.

[3] J. Diaz-De-Arcaya, J. López-De-Armentia, R. Miñón, I. L. Ojanguren, and A. I. Torre-Bastida, "Large language model operations (llmops): Definition, challenges, and lifecycle management," in *2024 9th International Conference on Smart and Sustainable Technologies (SpliTech)*. IEEE, 2024, pp. 1–4.

[4] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems." [Online]. Available: http://arxiv.org/pdf/2310.03533v4

[5] T. Chen, "Challenges and opportunities in integrating llms into continuous integration/continuous deployment (ci/cd) pipelines," in *2024 5th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*. IEEE, 2024, pp. 364–367.

[6] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2339–2356.

[7] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, Lo David, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review." [Online]. Available: http://arxiv.org/pdf/2308.10620v6

[8] P. d. O. Santos, A. C. Figueiredo, P. Nuno Moura, B. Diirr, A. C. F. Alvim, and R. P. D. Santos, "Impacts of the usage of generative artificial intelligence on software development process," in *Proceedings of the 20th Brazilian Symposium on Information Systems*, R. M. de Castro, J. M. N. David, J. C. Marques, T. M. de Classe, V. Ströele, and W. A. F. Silva, Eds. New York, NY, USA: ACM, 2024, pp. 1–9.

[9] Z. Wang, C. Feng, L. Liu, G. Jiao, and P. Ye, "The application of llms in the analysis and modeling of software requirements," in *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 2024, pp. 1143–1153.

[10] A. C. Tricco, E. Lillie, W. Zarin, K. K. O'Brien, H. Colquhoun, D. Levac, D. Moher, M. D. Peters, T. Horsley, L. Weeks *et al.*, "Prisma extension for scoping reviews (prisma-scr): checklist and explanation," *Annals of internal medicine*, vol. 169, no. 7, pp. 467–473, 2018.

[11] A. Carrera-Rivera, W. Ochoa, F. Larrinaga, and G. Lasa, "How to conduct a systematic literature review: A quick guide for computer science research," *MethodsX*, vol. 9, p. 101895, 2022.

[12] T. Mahbub, D. Dghaym, A. Shankarnarayanan, T. Syed, S. Shapsough, and I. Zualkernan, "Can gpt-4 aid in detecting ambiguities, inconsistencies, and incompleteness in requirements analysis? a comprehensive case study," *IEEE Access*, vol. 12, pp. 171 972–171 992, 2024.

[13] B. Wei, "Requirements are all you need: From requirements to code with llms," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 2024, pp. 416–422.

[14] H. Ding, Z. Fan, I. Guehring, G. Gupta, W. Ha, J. Huan, L. Liu, B. Omidvar-Tehrani, S. Wang, and H. Zhou, "Reasoning and planning with large language models in code development," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, R. Baeza-Yates and F. Bonchi, Eds. New York, NY, USA: ACM, 2024, pp. 6480–6490.

[15] X. Xia, Z. Jin, M. Aiello, D. Zhang, G. Liang, and X. Hu, "Software service engineering in the era of large language models," in *2024 IEEE International Conference on Software Services Engineering (SSE)*. IEEE, 2024, p. xxiii.

[16] D. Russo, "Navigating the complexity of generative ai adoption in software engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–50, 2024.

[17] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, A. Roychoudhury, A. Paiva, R. Abreu, M. Storey, R. Hierons, and H. Madeira, Eds. New York, NY, USA: ACM, 2024, pp. 102–106.

[18] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[19] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, M. Christakis and M. Pradel, Eds. New York, NY, USA: ACM, 2024, pp. 1592–1604.

[20] I. Ozkaya, "Application of large language models to software engineering tasks: Opportunities, risks, and implications," *IEEE Software*, vol. 40, no. 3, pp. 4–8, 2023.

[21] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, M. d'Amorim, Ed. New York, NY, USA: ACM, 2024, pp. 185–196.

[22] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

[23] V. Bayrı and E. Demirel, "Ai-powered software testing: The impact of large language models on testing methodologies," in *2023 4th International Informatics and Software Engineering Conference (IISEC)*. IEEE, 2023, pp. 1–4.

[24] T. Chang, S. Chen, G. Fan, and Z. Feng, "A self-iteration code generation method based on large language models," in *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2023, pp. 275–281.

[25] N. S. Mathews and M. Nagappan, "Test-driven development and llm-based code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, V. Filkov, B. Ray, and M. Zhou, Eds. New York, NY, USA: ACM, 2024, pp. 1583–1594.

[26] M. Chernigovskaya, D. Walia, K. Neumann, A. Hardt, A. Nahhas, and K. Turowski, "Towards a standardized business process model for llmops," in *Proceedings of the 27th International Conference on Enterprise Information Systems - Volume 1: ICEIS*, INSTICC. SciTePress, 2025, pp. 856–866.

[27] T. D. Viet and K. Markov, "Using large language models for bug localization and fixing," in *2023 12th International Conference on Awareness Science and Technology (iCAST)*. IEEE, 2023, pp. 192–197.

[28] N. Alshahwan, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Assured offline llm-based software engineering," in *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering*. New York, NY, USA: ACM, 2024, pp. 7–12.

[29] Y. Huang, Y. Chen, X. Chen, J. Chen, R. Peng, Z. Tang, J. Huang, F. Xu, and Z. Zheng, "Generative software engineering." [Online]. Available: http://arxiv.org/pdf/2403.02583v2

[30] M. Coutinho, L. Marques, A. Santos, M. Dahia, C. França, and R. de Souza Santos, "The role of generative ai in software development productivity: A pilot case study," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, B. Adams, T. Zimmermann, I. Ozkaya, D. Lin, and J. M. Zhang, Eds. New York, NY, USA: ACM, 2024, pp. 131–138.

[31] T. Şimşek, Ç. Gülşeni, and G. A. Olcay, "The future of software development with genai: Evolving roles of software personas," *IEEE Engineering Management Review*, pp. 1–8, 2024.

[32] A. Rajbhoj, A. Somase, P. Kulkarni, and V. Kulkarni, "Accelerating software development using generative ai: Chatgpt case study," in *Proceedings of the 17th Innovations in Software Engineering Conference*, S. K. Chakrabarti, A. Rastogi, S. Ghosh, R. Komondoor, R. K. Medicherla, L. Kumar, and S. Godboley, Eds. New York, NY, USA: ACM, 2024, pp. 1–11.

[33] A. Nguyen-Duc, P. Abrahamsson, and F. Khomh, *Generative AI for Effective Software Development*. Cham: Springer Nature Switzerland, 2024.