Leveraging Large Language Models in the Software Development Lifecycle: Opportunities and Challenges

Jasdeep Singh Bhalla[1], Mansimar Kaur Jodhka[2]
GoDaddy Inc., United States[1]
University of Southern California, United States[2]

***Abstract*—Large Language Models (LLMs) are increasingly integrated into software engineering workflows, yet existing studies provide fragmented or domain-specific examinations of their impact. This survey aims to systematically analyze how LLMs influence the Software Development Lifecycle (SDLC) end-to-end, identifying capabilities, limitations, risks, and emerging opportunities. We review 147 publications from 2017–2025 across ACM Digital Library, IEEE Xplore, ACL Anthology, and arXiv using predefined inclusion and exclusion criteria. Unlike prior surveys that focus narrowly on code generation or testing, this work provides an SDLC-wide synthesis supported by empirical benchmarks, industrial evidence, and a unified taxonomy mapping LLM capabilities to each phase of development. We further examine technical risks including hallucinations, dataset governance, robustness, security vulnerabilities, and auditability. The goal of this survey is to consolidate fragmented knowledge, highlight practical adoption challenges, and outline future research directions essential for building trustworthy, scalable, and effective LLM-enabled software engineering systems.**

***Keywords*—Large Language Models (LLMs); Software Development Lifecycle (SDLC); AI-assisted software engineering; automated code generation; software testing; software architecture; DevOps automation**

I. Introduction

Large Language Models (LLMs) such as GPT-4, Llama-3, and DeepSeek-Coder have introduced unprecedented automation and reasoning capabilities into software engineering. Developers now use LLMs for code generation, test creation, debugging assistance, documentation, architectural exploration, and deployment planning. As the industry rapidly integrates these tools into real-world systems, understanding their capabilities, limitations, and implications across the entire Software Development Lifecycle (SDLC) has become increasingly critical.

Despite the pace of adoption, existing literature remains fragmented. Some surveys focus primarily on code generation, others on testing, documentation, or developer productivity, leaving the full SDLC picture incomplete. Furthermore, few studies provide a unifying taxonomy or systematically compare LLM capabilities across multiple SDLC phases using empirical benchmarks. Industry reports highlight impressive productivity gains, yet academic literature also documents challenges such as hallucinations, security vulnerabilities, and difficulties integrating LLMs into large codebases or long-context reasoning tasks. This creates a clear need for a comprehensive survey that consolidates insights across domains, evaluates supporting evidence, and critically analyzes risks.

A. Research Gap and Motivation

Current surveys exhibit three key limitations:

1) Fragmented coverage: Existing reviews examine isolated aspects of the SDLC—such as code generation or requirements—but no survey systematically maps LLM capabilities across all SDLC phases.

2) Lack of unified taxonomy: Prior work rarely provides a structured framework that connects LLM tasks, techniques, risks, and empirical results to specific SDLC processes.

3) Insufficient evaluation perspective: Many surveys describe applications without synthesizing evidence from benchmark datasets, industrial deployments, or comparative performance analyses.

Given these gaps, a holistic and methodologically grounded SDLC-wide review is needed—one that not only describes what LLMs can do, but critically evaluates how well they perform, what risks they introduce, and where future work is necessary.

A detailed comparison of this survey with prior work is presented in Table III, highlighting gaps in SDLC coverage, taxonomy completeness, and empirical rigor.

B. Paper Scope and Organization

This survey consolidates insights from 147 peer-reviewed and high-impact sources published between 2017 and 2025. The paper is organized as follows:

- Section II presents background concepts and related work.
- Section III introduces the survey methodology, inclusion/exclusion criteria, and search strategy.
- Section IV proposes a unified taxonomy of LLM capabilities across SDLC phases.
- Section V analyzes LLM applications in requirements engineering, design, implementation, testing, deployment, and maintenance.
- Section VI examines risks and limitations including hallucinations, robustness, dataset governance, and security vulnerabilities.

- Section VII outlines future research opportunities and open challenges.
- Section VIII concludes with a synthesis of findings and implications.

This structured approach ensures clarity, enables reproducibility, and provides a coherent understanding of LLMs' role in modern software engineering.

C. Contributions

This work provides a comprehensive and SDLC-wide examination of Large Language Model (LLM) applications in software engineering. Unlike prior surveys that focus primarily on code generation, testing, or developer productivity, this survey offers a unified and methodologically rigorous perspective across all phases of the Software Development Lifecycle (SDLC). The major contributions of this paper are as follows:

1) An SDLC-wide unified taxonomy: We propose the first consolidated taxonomy that maps LLM capabilities, tasks, risks, and limitations across all major SDLC phases: requirements engineering, architecture and design, implementation, testing, deployment, maintenance, and security.

2) A Systematic review of 147 publications (2017–2025): We conduct a structured survey across ACM DL, IEEE Xplore, ACL Anthology, and arXiv using predefined search keywords, inclusion/exclusion criteria, and quality filters. Our methodology promotes reproducibility and grounded analysis.

3) Comparative synthesis across models and benchmarks: We evaluate and synthesize empirical performance results of popular LLMs (GPT-4, Llama-3, CodeLlama, DeepSeek-Coder, StarCoder) across widely used software engineering benchmarks including HumanEval, MBPP, SWE-Bench, BigCodeBench, and CodeXGLUE.

4) Analysis of practical adoption challenges: We identify real-world integration issues including hallucinations, misalignment, long-context reasoning limitations, dataset governance, licensing and IP concerns, and architectural inconsistency challenges.

5) A Risk and governance framework: We examine security threats (prompt injection, dependency confusion), robustness issues, model drift, cost-performance trade-offs, and data governance constraints relevant to enterprise adoption.

6) A Research roadmap for future work: Based on existing gaps, we propose future research directions including neuro-symbolic hybrids, agentic LLMs for end-to-end SDLC automation, safe-by-design AI coding tools, verified LLMs, long-context architectures, and multi-model ensemble reasoning.

II. Background and Related Work

Integration of Large Language Models (LLMs) in software engineering has garnered significant attention in recent years [20], with several studies and tools exploring their potential to enhance various phases of the Software Development Life Cycle (SDLC).

One notable example is OpenAI's Codex [14], which demonstrated impressive capabilities in translating natural language into source code. Codex helps developers write boilerplate code, suggest code improvements, and generate code snippets, providing significant time-saving benefits. Additionally, GitHub Copilot [5], powered by Codex, has become a popular tool integrated into development environments, offering auto-completion, real-time suggestions, and code snippets across a wide range of programming languages.

Recent studies have explored the potential of Large Language Models (LLMs) in software development, particularly in academic settings. While general-purpose LLMs are not primarily designed for code generation, they have shown promising results in assisting with software engineering tasks. One such study involved 214 students working in teams, where LLMs were integrated into their development toolchain. This study analyzed AI-generated code, the prompts used for code generation, and the human intervention needed to integrate the code into the existing codebase. Additionally, a perception study was conducted to understand the usefulness and challenges of LLMs from the students' perspective. Findings suggest that LLMs are particularly beneficial in the early stages of software development, aiding in the generation of foundational code structures, syntax correction, and error debugging. The study emphasizes the importance of preparing students for effective human-AI collaboration, offering valuable insights into how LLMs can enhance the productivity of software engineering students [16].

Jiang et al. (2024) present a comprehensive survey on Large Language Models (LLMs) for code generation, a rapidly growing field with significant implications for software development. It addresses the gap in the literature regarding a systematic review of LLMs specifically for code generation. It introduces a taxonomy to categorize and analyze advancements in this domain, covering data curation, model performance, ethical concerns, environmental impact, and real-world applications. The authors provide a historical overview and empirical comparisons using benchmarks like HumanEval, MBPP, and BigCodeBench, evaluating LLM performance across varying difficulty levels and task types. The study also identifies challenges and opportunities in bridging academic research with practical software development needs [9].

In the context of software testing, LLMs have shown considerable potential for automating key aspects such as test case creation and defect detection, leveraging machine learning and natural language processing. A comparative study by Boukhlif et al. (2024) examines various LLMs used in software testing, focusing on how they interact with, fine-tuning methods, and prompt engineering. The study provides valuable insights into the technologies and testing types that can be automated with LLMs, helping researchers and industry professionals select the most effective models for their testing needs. This research adds to the growing understanding of how LLMs can enhance the software testing process, offering guidance on their application in real-world scenarios [1].

Krishna et al. (2024) explore the potential of Large Language Models (LLMs) in the creation of Software Requirements Specifications (SRS), a key document in software development. Their study evaluates the ability of GPT-4 and CodeLlama to generate coherent, structured, and accurate drafts of an SRS for a university club management system. The results indicate that LLMs can match the output quality of an entry-level software engineer, producing complete and consistent drafts. Additionally, GPT-4 demonstrated the capa-

bility to identify issues and suggest improvements in existing SRS documents, while CodeLlama's performance in validation tasks was less effective. Their experiments also revealed that LLMs could significantly reduce the time required to draft SRS documents, ultimately enhancing productivity in software development [11].

III. Methodology

This section outlines the systematic methodology adopted for conducting this survey. Following established guidelines for evidence-based software engineering reviews, we designed a multi-stage process covering search strategy, screening, quality assessment, and data extraction to ensure reproducibility and rigor.

A. Search Strategy

We queried four major digital libraries between 2017–2025:

- ACM Digital Library
- IEEE Xplore
- ACL Anthology
- arXiv.org

Search strings combined LLM-related and software-engineering-related terms:

("large language model" OR LLM OR GPT OR Codex OR "code generation" OR "AI agents")

AND

("software engineering" OR SDLC OR requirements OR testing OR debugging OR deployment OR DevOps)

B. Inclusion and Exclusion Criteria

We included studies that met the following criteria:

- Inclusion Criteria
  - Published 2017–2025
  - Focused on LLM applications relevant to SDLC
  - Peer-reviewed or widely cited preprints (¿ 50 citations)
  - Provided empirical, conceptual, or methodological insights
- Exclusion Criteria
  - Non-English publications
  - Blog posts, opinion essays, or editorials
  - Papers without evaluation, case studies, or technical content

C. Screening Process

A three-stage filtering process was applied:

- Initial Retrieval: 1,182 papers identified.
- Title/Abstract Screening: 384 papers shortlisted.
- Full-Text Review: 147 papers selected.

A summary of the three-stage filtering and inclusion results is provided in Table I.

TABLE I. Screening Summary

Stage	Count
Initial Papers Retrieved	1182
After Title/Abstract Screening	384
Full-Text Assessed	221
Included in Final Review	147

D. Data Extraction and Coding

For each paper, we extracted:

- SDLC phase addressed
- Task type (requirements, code gen, testing, DevOps, etc.)
- Dataset or benchmark used
- Model type (GPT-4, Llama-3, Codex, etc.)
- Risks, limitations, or open challenges identified

Two researchers independently coded papers and resolved disagreements through discussion.

E. Quality Assessment

We evaluated studies using:

- Clarity of methodology
- Empirical rigor
- Relevance to SDLC
- Replicability of findings

Only papers scoring greater than or equal to 3/5 in quality assessment were included.

F. Research Questions

This review is guided by the following research questions (RQs):

- RQ1: What capabilities do Large Language Models (LLMs) demonstrate across different phases of the Software Development Lifecycle (SDLC)?
- RQ2: How do state-of-the-art LLMs perform on empirical benchmarks relevant to software engineering tasks?
- RQ3: What risks, limitations, and practical challenges arise when integrating LLMs into software engineering workflows?
- RQ4: What open research gaps and future opportunities exist for advancing LLM-enabled software engineering?

IV. Unified SDLC Taxonomy for LLM Applications

To provide a structured lens for analyzing LLM applications across the Software Development Lifecycle (SDLC), we introduce a unified taxonomy synthesizing tasks, capabilities, risks, and evidence from 147 publications. This taxonomy organizes LLM contributions into seven interconnected SDLC phases: Requirements, Design, Implementation, Testing, Deployment, Maintenance, and Security.

A. Taxonomy Overview

Our taxonomy consists of four dimensions:

- SDLC Phase: Requirements, Architecture, Implementation, Testing, Deployment, Maintenance, Security.
- LLM Capability Type: Generation, Summarization, Reasoning, Classification, Retrieval-Augmentation, Dialogue.
- Task Category: Tasks such as SRS drafting, UML generation, code generation, bug fixing, test generation, IaC synthesis, threat analysis, and log summarization.
- Risk Profile: Hallucination, brittleness, bias, security risk, data governance, IP leakage, misalignment.

The hierarchical organization of these relations is illustrated in Fig. 1, which presents the unified SDLC taxonomy used throughout this survey.

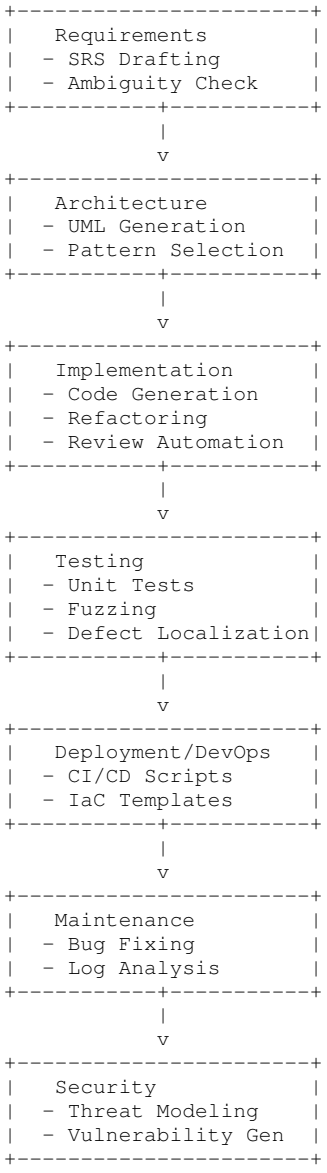

Fig. 1. Unified SDLC taxonomy for LLM-Driven software engineering.

B. Taxonomy Contributions

This taxonomy enables:

- A consistent mapping from LLM capability to SDLC activity
- Cross-phase comparison of LLM maturity levels
- Identification of gaps in research and industry adoption
- Systematic evaluation of risks associated with LLM usage

C. Taxonomy Structure

Table IV (already included later) operationalizes this taxonomy by mapping tasks to phases and benchmarks.

1) Advantages over existing methods: Unlike prior surveys that examine LLMs within isolated tasks such as code generation or testing, our approach provides an SDLC-wide analytical framework that unifies capabilities, risks, empirical evidence, and cross-phase interactions. Existing reviews do not integrate benchmark performance, industrial case studies, and security considerations into a single taxonomy. Our methodology therefore offers broader coverage, deeper comparative analysis, and a more actionable foundation for both researchers and practitioners. This holistic structure enables clearer identification of gaps, limitations, and opportunities that are not visible when examining SDLC phases independently.

Table II summarizes empirical performance of representative LLMs across common software engineering benchmarks.

TABLE II. Performance of Popular LLMs on SE Benchmarks

Model	HumanEval	MBPP	SWE-Bench
GPT-4	67.0%	82.0%	38.1%
Llama-3-70B	62.2%	79.4%	21.7%
CodeLlama-34B	48.8%	53.4%	9.0%
StarCoder2-15B	36.0%	48.0%	5.2%
DeepSeek-Coder-33B	57.0%	71.0%	18.0%

V. Applications in the Software Development Lifecycle (SDLC)

This section explores the transformative role of Large Language Models (LLMs) in various stages of the SDLC, highlighting their impact on code generation, system architecture, requirements gathering, and beyond. It delves into the foundational architectures that power LLMs, such as GPT (Generative Pre-trained Transformer) series, BERT (Bidirectional Encoder Representations from Transformers), CodeBERT, RoBERTa,

TABLE III. COMPARISON OF OUR SURVEY WITH EXISTING SURVEYS ON LLMS IN SOFTWARE ENGINEERING

Survey	Focus Area	Covers Full SDLC?	Empirical Benchmarks	Taxonomy Provided?	Risk Analysis
Hou et al. (2024) [6]	General SE tasks	No	Limited qualitative results	Partial	Limited
Jiang et al. (2024) [9]	Code generation	No (Implementation only)	HumanEval, MBPP	Yes (Code-gen taxonomy)	Minimal
Boukhlif et al. (2024) [1]	Software testing	No (Testing only)	LLM-based testing examples	Yes (Testing taxonomy)	No
Rasnayaka et al. (2024) [16]	Student dev workflows	No	Real classroom data	No	No
Jahić and Sami (2024) [8]	Software architecture	No (Architecture only)	Anecdotal	No	No
This Survey (2025)	**Entire SDLC (Requirements → Security)**	**Yes**	**HumanEval, MBPP, SWE-Bench, BigCodeBench, CodeXGLUE**	**Yes (Unified SDLC taxonomy)**	**Yes (Security, governance, compliance)**

TABLE IV. MAPPING OF LLM CAPABILITIES ACROSS SDLC PHASES

SDLC Phase	LLM Capabilities	Representative Evidence / Benchmarks	Maturity Level
Requirements	SRS drafting, ambiguity detection, requirement classification, user-story expansion	GPT-4 outperforming baselines (Krishna et al. 2024)	Moderate
Architecture & Design	UML generation, architecture suggestions, pattern selection, tradeoff reasoning	ICSA-C 2024 studies, industrial case reports	Low–Moderate
Implementation	Code generation, auto-completion, refactoring, code translation	HumanEval, SWE-Bench, MBPP, BigCodeBench	High
Testing	Unit test generation, fuzzing, test oracles, defect localization	ChatUniTest (FSE 2024), ICST 2024 results	Moderate–High
Deployment	CI/CD generation, IaC templates, environment config synthesis	Srivatsa et al. 2024, DevOps case studies	Low–Moderate
Maintenance	Bug localization, patch generation, log summarization, regression analysis	SWE-Bench Leaderboard	Moderate
Security	Threat modeling, vulnerability summary, prompt-injection detection	EuroS&PW 2024, industrial LLM security frameworks	Low

and other models relevant to enhancing software development practices.

A. Requirements Gathering and Planning

LLMs significantly enhance early-stage software development tasks such as requirements elicitation and project planning. By processing natural language inputs from user stories, feature requests, and other documentation, LLMs can help teams extract key information and ensure clarity in project scope and requirements [15]. Additionally, LLMs analyze historical project data to create optimized project timelines, resource estimates, and risk assessments, improving the accuracy and efficiency of planning phases.

B. Cost Estimation

LLMs assist in cost management by analyzing cloud usage patterns and resource utilization data. They offer insights into optimizing cloud expenses, proposing budgeting strategies, and suggesting effective ways to manage resources like reserved instance planning and resource tagging, all of which contribute to more cost-effective cloud deployments.

C. Architecture and Design

LLMs are increasingly valuable in system architecture and design. They support the design process by offering architectural suggestions based on system requirements and constraints, assisting in identifying suitable patterns and components. LLMs also facilitate the generation of architectural documentation and design specifications, helping teams visualize and iterate on system design.

1) High-level system design: LLMs can assist in architectural design by providing insights, recommendations, and solutions for designing software systems. By analyzing natural language descriptions of system requirements, constraints, and objectives, LLMs can help identify suitable architectural patterns, components, and design decisions. Additionally, LLMs can facilitate the exploration of alternative architectural designs and the evaluation of their trade-offs, enabling architects to make informed decisions based on a holistic understanding of system requirements and design goals [8].

2) Low-level system design: LLMs can be used to enhance low-level system design by automating code generation that adheres to clean code principles and Object-Oriented Programming (OOP) practices. They can assist in writing modular, maintainable, and optimized code by suggesting well-structured functions, meaningful variable names, and applying design patterns such as encapsulation, inheritance, and polymorphism. LLMs can help in refactoring complex low-level code, improving readability, and identifying performance optimizations, which are crucial for system programming.

3) User Interface (UI) Design: LLMs can be a powerful tool in User Interface (UI) Design, helping designers and developers create intuitive, user-friendly interfaces more efficiently [13]. Here's how they can be used:

a) Generating UI components: LLMs can assist in generating code for common UI components (buttons, forms, etc.) in various front-end frameworks (e.g., React, Angular) based on natural language descriptions.

b) Design recommendations: LLMs can analyze existing UI designs and suggest improvements based on best practices, ensuring that the design is both visually pleasing and functional.

c) Automated prototyping: LLMs can generate mockups and wireframes from textual input, enabling rapid prototyping. By converting textual descriptions into visual elements, LLMs help speed up the iterative design process and provide a starting point for more detailed work.

d) UI testing automation: LLMs can generate test cases for UI components, ensuring that the design works as intended across different devices and screen sizes.

4) Data modeling: LLMs can aid in modeling and specifying system requirements, components, interfaces, and behaviors using natural language descriptions. By generating formal models, such as Unified Modeling Language (UML) diagrams, state diagrams, and sequence diagrams, LLMs can help visualize system designs and communicate them effectively to stakeholders.

a) Schema generation: LLMs can automatically generate database schemas or data models from textual descriptions or requirements. Developers can input a set of business logic or requirements, and LLMs can produce entity-relationship diagrams (ERDs) and relational models.

b) Data type inference: Given data descriptions, LLMs can suggest the appropriate data types (e.g., integer, string, boolean) and constraints (e.g., primary keys, foreign keys), ensuring data integrity.

c) Automated documentation: LLMs can generate comprehensive documentation for data models, providing explanations for tables, fields, relationships, and constraints.

5) API specifications: LLMs can assist in creating detailed specifications, including system requirements documents, interface specifications, and design documentation, ensuring clarity and consistency in system design artifacts [12].

a) API documentation generation: LLMs can automatically generate structured and clear API documentation, including endpoints, request parameters, responses, and error codes.

b) Endpoint design: LLMs can suggest appropriate API endpoints and methods (GET, POST, PUT, DELETE) based on business requirements, as well as naming conventions and data structures.

c) Request/Response format design: LLMs can recommend best practices for request and response formats (e.g., JSON, XML) and generate example payloads based on the API structure.

6) Integration of data modeling and APIs: LLMs can analyze data models and suggest how to expose data entities as APIs, ensuring seamless integration between the data and API layers. They can help verify that API endpoints correctly handle data and perform transformations, ensuring consistency between the data model and API.

D. Development and Implementation

Large Language Models (LLMs) have shown promise in aiding various aspects of code implementation, encompassing code generation, auto-completion, refactoring, and style enforcement. This section explores how LLMs can be leveraged in code implementation processes:

1) Code generation: LLMs can assist developers in generating code snippets, functions, or even entire programs based on natural language descriptions of desired functionality. By understanding the context and intent conveyed in the description, LLMs can produce syntactically correct code that aligns with the provided specifications. This capability can be particularly useful for prototyping, scaffolding, or automating repetitive coding tasks [9].

2) Code review: LLMs can be effectively utilized to assist in code reviews by analyzing code for potential bugs, inefficiencies, and adherence to coding standards. LLMs can automatically identify common programming errors, security vulnerabilities, and suggest improvements in code structure and logic. Additionally, they can provide detailed explanations and recommendations for refactoring, making the code more readable and maintainable. By comparing the code against best practices and known design patterns, LLMs can help developers ensure code quality and consistency. This not only speeds up the review process but also enhances the accuracy and depth of the feedback provided, complementing human reviewers in identifying subtle issues [6].

3) Refactoring: LLMs can aid developers in refactoring existing codebases by suggesting code transformations, optimizations, or restructuring based on natural language descriptions of desired changes. By understanding the semantics and relationships within the code, LLMs can provide actionable recommendations for improving code readability, performance, and maintainability. This can help developers refactor code more confidently and efficiently, leading to cleaner and more scalable codebases [7].

4) Documentation generation: LLMs can automatically generate documentation from source code by analyzing comments, function signatures, and code logic to produce detailed descriptions and usage examples. This capability extends to creating API documentation, where LLMs can interpret API contracts and annotations to draft comprehensive endpoint descriptions and parameter explanations. Additionally, LLMs can assist in drafting user manuals and setup guides by translating technical specifications into user-friendly language. They can also automate the creation of change logs and release notes by summarizing commit messages and version changes [4].

E. Testing and Validation

Large Language Models (LLMs) offer significant potential for enhancing various aspects of code testing, including test case generation, test script creation, and test oracle generation. This section explores how LLMs can be leveraged in code-testing processes:

1) Test case generation: LLMs can assist in automatically generating test cases based on natural language descriptions of software requirements, functionalities, and edge cases. By understanding the semantics and logic embedded in the descriptions, LLMs can generate diverse and comprehensive test cases that cover various scenarios, inputs, and outcomes [18]. This can help improve test coverage, identify corner cases, and ensure robustness and reliability in software applications [3].

2) Automated bug detection: LLMs can aid in automated bug detection by analyzing natural language descriptions of

reported issues, error messages, or bug reports. By understanding the context and symptoms described in the reports, LLMs can assist in identifying potential root causes, suggesting debugging strategies, or providing insights into possible solutions. This can help expedite the bug triage process, improve the accuracy of bug classification, and facilitate the timely resolution of software defects.

F. Deployment and Release Management

Large Language Models (LLMs) offer significant potential for streamlining various aspects of code deployment, including automation, continuous integration, continuous delivery, and release management. This section explores how LLMs can be leveraged in code deployment processes:

1) Automation of deployment tasks: LLMs can assist in automating repetitive deployment tasks, such as package installation, configuration management, and environment setup. By interpreting natural language descriptions of deployment requirements and procedures, LLMs can generate scripts or workflows that automate the deployment process across different environments and platforms. This can help reduce manual effort, minimize human errors, and accelerate the deployment cycle, thereby improving efficiency and reliability in software deployment [10].

2) Continuous Integration and Continuous Delivery (CI/CD): LLMs can facilitate continuous integration and continuous delivery (CI/CD) pipelines by generating configuration files, build scripts, and deployment pipelines based on natural language descriptions of CI/CD workflows. By understanding the dependencies, triggers, and stages involved in the CI/CD process, LLMs can help developers set up and customize CI/CD pipelines to automate code builds, testing, and deployment. This can enable faster feedback loops, shorter release cycles, and greater agility in software development and deployment practices.

3) Release management: LLMs can aid in release management by generating release notes, changelogs, and versioning schemes based on natural language descriptions of software changes and updates. By analyzing commit messages, issue descriptions, and release plans, LLMs can assist in summarizing and documenting the changes introduced in each software release. This can help improve transparency, communication, and collaboration among development teams, stakeholders, and end-users during the release process [2].

4) Infrastructure as Code (IaC): LLMs can support Infrastructure as Code (IaC) practices by generating infrastructure configuration files, provisioning scripts, and deployment manifests based on natural language descriptions of infrastructure requirements and specifications. By understanding the desired infrastructure topology, components, and configurations, LLMs can help automate the provisioning and deployment of infrastructure resources in cloud environments. This can improve consistency, scalability, and reproducibility in infrastructure management and deployment processes [19].

5) Monitoring and logging: LLMs can analyze system logs, performance metrics, and monitoring data to detect anomalies, performance bottlenecks, and security incidents. They can provide insights for system optimization and troubleshooting.

G. Security and Compliance

LLMs can enhance cybersecurity by assisting in threat detection, vulnerability assessment, and incident response. They analyze natural-language descriptions of security incidents, attack patterns, and vulnerabilities to identify suspicious activities and security weaknesses. By interpreting reports and advisories, LLMs help prioritize vulnerabilities based on severity and potential impact. Leveraging LLMs can significantly improve an organization's ability to detect, assess, and respond to security threats, ultimately enhancing its overall security posture [17].

VI. Challenges and Limitations

Despite the promising capabilities of Large Language Models (LLMs) in software development, their integration into the Software Development Life Cycle (SDLC) presents several limitations and challenges. While LLMs can significantly enhance productivity and automation in many SDLC phases, various technical and practical issues still hinder their widespread adoption.

1) Code accuracy and reliability: One of the primary challenges is the accuracy of the code generated by LLMs. While LLMs can produce syntactically correct and functional code, they often lack contextual understanding, leading to errors or code that doesn't meet specific functional requirements. This phenomenon, often referred to as "hallucination," occurs when LLMs generate plausible-sounding code that may be incorrect, inefficient, or insecure.

2) Lack of domain expertise: LLMs are trained on vast, general datasets and lack specific domain expertise. They may not understand the nuances of specialized domains such as healthcare, finance, or cybersecurity, leading to inaccurate or incomplete code generation.

3) Limited understanding of software context: LLMs generate code based on input prompts but often lack awareness of the broader software context, such as existing codebases, architectural constraints, or dependency management. This limitation can lead to code that does not fit well with existing systems, requiring developers to invest time in manually integrating and validating the generated code.

4) Bias in training data: LLMs are susceptible to bias present in their training data, which can lead to the generation of biased or unfair code. This can result in unintended consequences, such as reinforcing harmful stereotypes or excluding certain groups.

5) Data privacy and security concerns: The use of LLMs in software development introduces potential privacy and security risks. Since LLMs are trained on publicly available data, there is a risk that sensitive or proprietary information might be unintentionally embedded in the model, leading to data leakage.

6) Ethical and legal concerns: The use of LLMs in software development raises ethical and legal issues. For instance, LLMs may inadvertently reproduce proprietary or copyrighted code snippets, which can lead to intellectual property violations.

While current advancements have showcased the power of LLMs in generating code, a greater focus on mitigating risks, improving accuracy, and ensuring ethical use is necessary for these models to fully integrate into the SDLC and become a reliable tool for developers.

Unlike prior surveys, which limit their focus to isolated SDLC stages (e.g., code generation, testing, architecture), this work provides the first SDLC-wide comparative survey integrating academic evidence, industrial deployment reports, empirical benchmark results, and a comprehensive taxonomy that aligns LLM capabilities with the end-to-end software engineering process. This breadth, combined with a structured methodology and risk/governance analysis, positions this work as a uniquely holistic reference for researchers and practitioners.

VII. Future Directions and Opportunities

The future of Large Language Models (LLMs) in software engineering is poised to bring transformative changes across various aspects of the field. Key trends and future directions include:

A. Advancements in Code Analysis and Debugging

1) Automated code quality checks: LLMs can help detect common coding errors, anti-patterns, and violations of best practices. They can also enforce code style guides and formatting rules, ensuring consistency across projects.

2) Context-aware suggestions: LLMs can provide tailored feedback by understanding the project context, including the architecture and libraries in use. Based on this, they suggest optimized algorithms or efficient library methods suited to the existing codebase.

3) Automated bug detection and fixes: Future LLMs will offer more sophisticated bug detection and correction capabilities, analyzing code and logs to identify issues and suggest fixes. This would reduce manual debugging efforts and improve software reliability.

4) Predictive analytics: LLMs will leverage historical data and trends to predict potential issues and provide proactive recommendations. This could include predicting code failures, suggesting optimizations, and forecasting project risks.

5) Root cause analysis: Advanced LLMs will enhance root cause analysis by identifying underlying issues affecting software performance. They will analyze code dependencies, execution patterns, and historical data to provide actionable insights for resolving complex problems.

B. Advancements in Development Practices

1) Rapid prototyping and experimentation: LLMs enable developers to experiment with different languages and frameworks without needing deep expertise. This includes quickly prototyping solutions in multiple languages to find the best fit for a project and supporting hybrid approaches, such as combining Python for data processing with Go for high-performance services.

2) Improved legacy system modernization: LLMs assist in modernizing legacy systems by analyzing old code, identifying redundancies, and suggesting optimal migration strategies. They help preserve business logic during the transition and reduce time and resources needed for manual migration, ultimately offering cost savings.

C. Advancements in Cross-Domain Innovations

1) Cross-language code translation: LLMs can facilitate cross-language code translation, enabling efficient migration of legacy code (e.g., COBOL, PHP or Perl) to modern languages like Python, Java, and Go. They also support transitioning between ecosystems (e.g., JavaScript to TypeScript, Swift to Kotlin) and produce clean, well-documented code that adheres to best practices in the target language, reducing technical debt and ensuring compatibility.

2) Automated version upgrades: LLMs can assist in automating version upgrades by refactoring deprecated syntax, suggesting new language features or libraries for improved performance, and ensuring backward compatibility between different language versions, keeping codebases up-to-date and aligned with the latest advancements.

3) Cross-domain interoperability: LLMs can facilitate cross-domain interoperability by enabling systems in different languages to communicate through automated API integration, ensuring code standardization across projects, and generating multi-language documentation for global teams.

Overall, the future of LLMs in software engineering promises increased automation, enhanced efficiency, and more intelligent tools, revolutionizing how software is developed, maintained, and managed.

D. Limitations

This survey is limited by publication availability up to early 2025, potential search bias in digital libraries, and reliance on reported empirical results that may lack standardized evaluation. Future surveys may extend the dataset and incorporate longitudinal analyses as LLMs evolve rapidly.

VIII. Conclusion

This survey provides a systematic and comprehensive analysis of the role of Large Language Models across the Software Development Lifecycle. By synthesizing findings from 147 publications, we show how LLMs influence requirements engineering, architectural design, implementation, testing, deployment, and maintenance. In contrast to prior reviews that focus on isolated SDLC phases, this work offers an SDLC-wide taxonomy and evidence-based synthesis that highlight both the breadth and depth of LLM adoption in modern software engineering.

Our review demonstrates that LLMs deliver tangible benefits—accelerated prototyping, improved code quality, enhanced test generation, and more efficient DevOps workflows. At the same time, these capabilities introduce significant risks, including hallucinations, security vulnerabilities, dataset governance issues, and limitations in long-context and architecture-level reasoning. These challenges emphasize the need for careful evaluation and responsible integration of LLM-based

tools, particularly in safety-critical or large-scale production environments.

The findings of this survey point to several clear research priorities: increasing model robustness, enabling reliable long-context reasoning, improving transparency and auditability, securing LLM pipelines, and advancing hybrid neuro-symbolic techniques that combine machine learning with formal verification. Progress in these areas will be essential for realizing trustworthy, scalable, and enterprise-ready LLM-enabled software engineering systems.

By addressing the research gaps identified in the introduction—fragmented coverage, lack of unified taxonomies, and limited empirical synthesis—this survey provides a cohesive reference framework for the software engineering and AI communities. The taxonomy, comparative analyses, and risk assessments presented here aim to guide future research directions, support tool builders, and assist practitioners in integrating LLMs responsibly and effectively into real-world development workflows.

REFERENCES

[1] M. Boukhlif, N. Kharmoum, and M. Hanine, "LLMs for Intelligent Software Testing: A Comparative Study," in *Proc. 7th Int. Conf. on Networking, Intelligent Systems and Security (NISS '24)*, ACM, 2024, pp. 1–8. doi: 10.1145/3659677.3659749.

[2] T. Chen, "Challenges and Opportunities in Integrating LLMs into Continuous Integration/Continuous Deployment (CI/CD) Pipelines," in *2024 5th Int. Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, IEEE, Nanjing, China, 2024. doi: 10.1109/AINIT61980.2024.10581784.

[3] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: A Framework for LLM-Based Test Generation," in *FSE 2024: Companion Proc. of the 32nd ACM Int. Conf. on the Foundations of Software Engineering*, ACM, 2024, pp. 572–576. doi: 10.1145/3663529.3663801.

[4] S. S. Dvivedi, V. Vijay, S. L. R. Pujari, S. Lodh, and D. Kumar, "A Comparative Analysis of Large Language Models for Code Documentation Generation," in *Proc. 1st ACM Int. Conf. on AI-Powered Software (AIware 2024)*, ACM, 2024, pp. 65–73. doi: 10.1145/3664646.3664765.

[5] GitHub, GitHub Copilot, 2024. [Online]. Available: https://github.com/features/copilot

[6] X. Hou *et al.*, "Large Language Models for Software Engineering: A Systematic Literature Review," *arXiv preprint arXiv:2308.10620*, Apr. 2024. doi: 10.48550/arXiv.2308.10620.

[7] R. A. Husein, H. Aburajouh, and C. Catal, "Large language models for code completion: A systematic literature review," *Computer Standards & Interfaces*, vol. 92, p. 103917, Mar. 2025. doi: 10.1016/j.csi.2024.103917.

[8] J. Jahić and A. Sami, "State of Practice: LLMs in Software Engineering and Software Architecture," in *Proc. 2024 IEEE 21st Int. Conf. on Software Architecture Companion (ICSA-C)*, Hyderabad, India, IEEE, 2024. doi: 10.1109/ICSA-C63560.2024.00059.

[9] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," *arXiv preprint arXiv:2406.00515*, Jun. 2024. doi: 10.48550/arXiv.2406.00515.

[10] H. Jin *et al.*, "From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future," *arXiv preprint arXiv:2408.02479*, 2024. doi: 10.48550/arXiv.2408.02479.

[11] M. Krishna, B. Gaur, A. Verma, and P. Jalote, "Using LLMs in Software Requirements Specifications: An Empirical Evaluation," *arXiv:2404.17842 [cs.SE]*, 2024. doi: 10.48550/arXiv.2404.17842. (Accepted to RE@Next! at IEEE RE 2024, Reykjavik, Iceland).

[12] K. Lazar *et al.*, "SpeCrawler: Generating OpenAPI Specifications from API Documentation Using Large Language Models," *arXiv preprint arXiv:2402.11625*, 2024. doi: 10.48550/arXiv.2402.11625.

[13] Y. Lu *et al.*, "UI Layout Generation with LLMs Guided by UI Grammar," *arXiv preprint arXiv:2310.15455*, 2023. doi: 10.48550/arXiv.2310.15455. (ICML 2023 Workshop on AI and HCI).

[14] OpenAI, "OpenAI Codex," 2024. [Online]. Available: https://openai.com/index/openai-codex/

[15] I. Ozkaya, "Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications," *IEEE Software*, vol. 40, no. 3, pp. 4–8, May–Jun. 2023. doi: 10.1109/MS.2023.3248401.

[16] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, "An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project," in *LLM4Code '24: Proc. 1st Int. Workshop on Large Language Models for Code*, ACM, 2024, pp. 111–118. doi: 10.1145/3643795.3648379.

[17] A. Salman, S. Creese, and M. Goldsmith, "Leveraging Large Language Models for Cybersecurity Compliance," in *Proc. 2024 IEEE EuroS&PW*, Vienna, Austria, IEEE, 2024. doi: 10.1109/EuroSPW61312.2024.00061.

[18] R. Santos, I. Santos, C. Magalhaes, and R. de S. Santos, "Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing," in *2024 IEEE Conf. on Software Testing, Verification, and Validation (ICST)*, IEEE, May 2024. doi: 10.1109/ICST.2024.00012.

[19] K. G. Srivatsa, S. Mukhopadhyay, G. Katrapati, and M. Shrivastava, "A Survey of Using Large Language Models for Generating Infrastructure as Code," *arXiv preprint arXiv:2404.00227*, 2024. doi: 10.48550/arXiv.2404.00227.

[20] H. Zhang and H. Shao, "Exploring the Latest Applications of OpenAI and ChatGPT: An In-Depth Survey," *Computer Modeling in Engineering & Sciences*, vol. XXX, no. X, 2023. doi: 10.32604/cmes.2023.030649.