Large-Scale Integration of Large Language Models into Software Engineering: Toward a Comprehensive Framework for Testing, Evaluation, and Deployment

[1] Dr. Arjun Mehta
[1] Department of Computer Science, University of Edinburgh, UK

Received: 01th Nov 2025 | Received Revised Version: 20th Nov 2025 | Accepted: 01th Dec 2025 | Published: 17th Dec 2025

Volume 07 Issue 12 2025 |

Abstract

With the rapid evolution and proliferation of Large Language Models (LLMs) in natural language processing, researchers and practitioners increasingly explore their potential in software engineering domains such as code generation, automated testing, and deployment workflows. This article presents a comprehensive conceptual analysis integrating insights from recent surveys and empirical studies to propose a unified framework for effectively leveraging LLMs across the software development lifecycle. Drawing on major works, including the broad survey of LLM architectures and capabilities (Zhao et al., 2024), the domain-specific evaluation of code generation tasks (Chen et al., 2024), and in-depth analyses of software testing with LLMs (Wang et al., 2024; Fan et al., 2023; Hou et al., 2024), this research systematically synthesizes existing findings, identifies critical gaps, and outlines a structured methodology to address key challenges. The findings highlight substantial variability in evaluation standards, a lack of robust testing pipelines tailored to LLM-generated code, deployment scalability constraints, and limited consensus on best practices. The proposed framework encompasses taxonomy, evaluation guidelines, testing strategies, and deployment infrastructure recommendations. This framework aims to guide future empirical research, industrial adoption, and standardization efforts in integrating LLM-powered tools into software engineering. The article concludes by discussing limitations and suggesting directions for future work, including empirical validation, benchmarking protocols, and governance considerations.

Keywords: large language models, software engineering, code generation, automated testing, deployment framework, evaluation metrics, scalability

Cite This Article: Dr. Arjun Mehta. (2025). Large-Scale Integration of Large Language Models into Software Engineering: Toward a Comprehensive Framework for Testing, Evaluation, and Deployment. The American Journal of Interdisciplinary Innovations and Research, 7(12), 61–67. Retrieved from https://theamericanjournals.com/index.php/tajiir/article/view/7085

1. Introduction

In recent years, Large Language Models (LLMs) have revolutionized the field of natural language processing and understanding. Their capacity to model rich linguistic patterns, generate coherent text, and perform various downstream tasks has spurred rapid adoption beyond conventional language tasks. Among these emerging applications, the use of LLMs in software engineering has garnered substantial attention. The idea of leveraging LLMs to assist or even automate tasks such as code generation, test case generation, documentation, and other development activities promises to reshape traditional software development paradigms. Indeed, several recent comprehensive surveys have attempted to map the landscape of LLMs and their applications in software engineering (Zhao et al., 2024; Raiaan et al., 2024; Hou et al., 2024; Fan et al., 2023).

On the one hand, the broader work by Zhao et al. (2024) presents a panoramic overview of LLM architectures, training, capabilities, and limitations. On the other, domain-specific investigations such as Chen et al. (2024) and Wang et al. (2024) examine the performance and challenges of LLMs in code generation and software testing contexts. Despite this burgeoning literature, there remains a conspicuous absence of a unified, holistic framework that ties together evaluation, testing, deployment, and operationalization of LLMs in real-world software engineering pipelines. In particular, existing works often treat code generation evaluation, test generation, and deployment scalability as disjoint problems, failing to address the integrated nature of software development environments. Moreover, while conventional software testing and engineering methodologies are well-established (ISO/IEC/IEEE 24765:2017), their adaptation to the unique characteristics of LLM-generated artifacts remains underexplored.

This article therefore addresses the following core research problem: How can we systematically integrate LLM-based tools into conventional software engineering workflows through a unified framework that addresses evaluation, testing, and deployment, while acknowledging the unique challenges of LLM-generated code and infrastructure demands? The absence of standardization, the diversity of evaluation metrics, and the lack of robust, scalable testing/integration platforms constitute critical gaps. These gaps hinder both academic reproducibility and industrial adoption of LLM-based tools.

To address this problem, this article builds a comprehensive conceptual framework by synthesizing evidence and perspectives across multiple seminal and recent works. The framework comprises: (1) a taxonomy of LLM roles in software engineering; (2) evaluation guidelines and standardized metrics; (3) integrated testing strategies combining traditional software testing paradigms with LLM-specific considerations; and (4) deployment and infrastructure recommendations mindful of scalability, resource management, and governance. By doing so, this article advances the discourse beyond isolated case studies or narrow surveys, offering a holistic roadmap for both researchers and practitioners.

2. Methodology

This work is based on a systematic synthesis of existing peer-reviewed literature and preprints focusing on LLMs and their applications in software engineering. The methodology comprises three stages: selection of sources, thematic coding, and framework construction.

First, we compiled a set of foundational and recent publications that directly address LLMs in general and their use in software engineering, testing, and code generation. Core works include broad surveys of LLMs (Zhao et al., 2024; Raiaan et al., 2024; Hou et p al., 2024), specialized studies on software engineering applications (Fan et al., 2023; Wang et al., 2024), and focused analyses on code generation evaluation (Chen et al., 2024). To contextualize LLM adoption within established software engineering practices, traditional references such as the standard vocabulary provided by ISO/IEC/IEEE 24765:2017 were included, along with classical investigations on software testing techniques (Mayeda & Andrews, 2021; Lonetti & Marchetti, 2018; Clark, Walkinshaw & Hierons, 2021). Additionally, to address deployment and infrastructure concerns, a few works that discuss scalable test platforms or cloud-based frameworks were considered (Chandra, 2025; Vasireddy et al., 2023; Pogiatzis & Samakovitis, 2020; Borra, 2024; Zhou et al., 2025), albeit more peripherally owing to their variable relevance.

Second, we conducted thematic coding of the identified literature. Each paper was analyzed to extract (a) the role(s) LLMs play in software engineering (e.g., code generation, test generation, documentation), (b) evaluation methodologies and metrics employed, (c) identified challenges, limitations, or open problems, (d) infrastructure or deployment considerations, and (e) proposed solutions or recommendations. Using these codes, we constructed a consolidated thematic map that reveals overlaps, divergences, and under-studied intersections.

Finally, based on the thematic map, we formulated a structured framework integrating taxonomy, evaluation, testing, and deployment. The framework is not empirical in itself but conceptual: it serves as a blueprint for future empirical validation and tool development.

Throughout, we strictly rely on information contained within the selected literature; we do not draw from external empirical experiments or data beyond what is available in the references. As such, the framework represents a reasoned, evidence-based synthesis, not a new empirical study.

3. Results

The synthesis yields four major outcomes: a taxonomy of LLM roles in software engineering, an evaluation guideline, an integrated testing strategy, and deployment/infrastructure recommendations.

Taxonomy of LLM Roles in Software Engineering

Our analysis reveals that LLMs currently contribute to software engineering through multiple, overlapping roles. These can be broadly categorized as follows:

● Code Generation and Augmentation: LLMs are used to generate new code, suggest code completions, refactor existing code, or add documentation comments. This is the most widely studied application. According to Chen et al. (2024), evaluations of LLM output in code generation tasks vary widely in metrics and granularity.

● Automated Test Case Generation: LLMs can generate test cases, including unit tests, integration tests, or edge-case scenarios. This use case has been explored by Wang et al. (2024) among others.

● Requirement Specification and Documentation: Some studies describe using LLMs to draft requirements, generate design documents, or convert natural-language specifications into formal representations. While less intensely studied, this role emerges in the broad surveys (Zhao et al., 2024; Raiaan et al., 2024).

● Code Review and Static Analysis Aid: LLMs can assist developers by reviewing code, flagging potential bugs, or suggesting improvements. This was highlighted in domain analyses of LLM applications in software engineering (Hou et al., 2024; Fan et al., 2023).

● Deployment and DevOps Assistance: Emerging works (e.g., Chandra, 2025) propose using LLMs to assist in deployment scripting, CI/CD automation, or configuration generation, although empirical evidence remains limited.

This taxonomy underscores that LLMs are not confined to a single narrow task—but span multiple phases of the software development lifecycle, from planning through coding to deployment and maintenance.

Evaluation Guidelines and Standardized Metrics

A striking result is the lack of consensus across studies regarding how to evaluate LLM-generated code and artifacts. We observed considerable heterogeneity in metrics, evaluation processes, and benchmarking. Some of the key observations:

● Diversity of Metrics: In code generation tasks, some studies measure correctness purely via compilation and execution (e.g., pass/fail tests), while others employ human evaluation for readability, maintainability, or style (Chen et al., 2024). For test generation, criteria include code coverage, fault detection capability, and even human assessment of test quality (Wang et al., 2024; Fan et al., 2023).

● Lack of Benchmark Standardization: While some authors attempt benchmark suites, the content, complexity, and domain coverage vary widely between studies. This makes comparison across different LLMs or configurations difficult.

● Limited Focus on Maintainability and Security: Few works systematically examine nonfunctional aspects such as code security, technical debt, maintainability, or long-term readability.

● Absence of Governance or Compliance Metrics: As LLM-generated artifacts become part of production codebases, issues like licensing, compliance, and copyright no longer remain theoretical. Yet none of the surveyed works propose metrics or processes to handle such concerns.

From this synthesis, we propose preliminary evaluation guidelines:

1. Multi-dimensional Evaluation: Any evaluation of LLM-generated code should include at least three dimensions: functional correctness (compilation, execution, test pass rate), code quality (readability, maintainability, style), and nonfunctional concerns (security, compliance, modularity).

2. Hybrid Benchmarking Approach: Combine automated benchmark suites with human expert review. Benchmarks should capture code complexity, domain diversity, and real-world relevance.

3. Longitudinal Assessment: Beyond one-time evaluation, code should ideally be monitored for maintenance cost, refactoring effort, bug occurrence, and team adoption over time.

4. Transparent Reporting: Researchers should report not only success rates but also failure modes, common error patterns, and instances where human intervention was required.

Integrated Testing Strategy

Integrating traditional software testing methodologies with LLM-specific concerns yields several insights. Traditional software testing techniques—unit testing, integration testing, regression testing—form a stable foundation (ISO/IEC/IEEE 24765:2017; Mayeda & Andrews, 2021; Lonetti & Marchetti, 2018). However, LLM-generated code introduces unique risks: unpredictable semantics, variability in code style, potential hidden vulnerabilities, and varying code structure. Our synthesis suggests the following integrated testing strategy:

● Initial Automated Testing: Immediately after generation, code must undergo standard automated tests including compilation, unit tests (if available), integration tests, and static analysis. This is no different from human-written code.

● LLM-Specific Sanitization Pass: Treat LLM-generated code as potentially "untrusted input," running additional static analysis and linting; scanning for known vulnerability patterns; checking for compliance with coding standards; verifying license headers or dependencies. This sanitization acknowledges that LLM outputs may inadvertently introduce security flaws or license violations.

● Fuzz Testing and Robustness Checks: Where possible, generate randomized input sets to test edge cases, ensure no unexpected behavior, and stress-test boundary conditions.

● Human Code Review: Especially important for complex or critical modules — human developers should review architecture, logic flow, readability, and compliance with project-specific conventions.

● Regression Testing and Monitoring Post-Integration: Once merged into the codebase, subsequent changes should trigger regression tests to ensure LLM-generated code continues to behave as expected under evolving dependencies and usage patterns.

This integrated strategy aims to combine the strengths of traditional testing with additional safeguards tailored to the particular uncertainties of LLM-generated artifacts.

Deployment and Infrastructure Recommendations

A frequent omission in the literature is consideration of deployment, scalability, and infrastructure — critical factors for productionizing LLM-enhanced software development tools. While specialized works are limited, a few promising directions emerge. For example, Chandra (2025) discusses design and implementation of scalable test platforms for LLM deployments; Vasireddy et al. (2023) explore load balancing in containerized orchestration contexts; and Pogiatzis & Samakovitis (2020) illustrate serverless ETL pipeline design on cloud platforms. Drawing from these, we outline a set of infrastructure recommendations:

1. Containerization and Orchestration: Use container-based environments (e.g., Docker) orchestrated via Kubernetes or similar for modularity, scalability, and portability. This facilitates isolated sandboxing of LLM inference workloads, test execution, and deployment tasks.

2. Horizontal Scaling with Load Balancing: To handle demand spikes — e.g., many developers requesting code generation or test generation concurrently — adopt load balancing and dynamic scaling strategies to distribute inference and testing workloads across multiple nodes (Vasireddy et al., 2023).

3. Serverless or Microservices-Based Architecture for Pipeline Components: Decompose the entire workflow — code generation, static analysis, test execution, result reporting — into microservices or serverless functions for scalability, maintainability, and ease of integration (Pogiatzis & Samakovitis, 2020).

4. CI/CD Integration and Automated Feedback Loops: Embed LLM-powered generation and testing pipelines into continuous integration/continuous deployment (CI/CD) workflows. Automated feedback (test pass/fail, linting results, security scan results) should be returned to developers promptly, enabling iterative refinement.

5. Monitoring, Logging, and Governance: Maintain detailed logs of LLM prompts, generated code snippets, test results, and deployment metadata. This supports traceability, compliance audits, and root-cause analysis for failures or unexpected behavior. Additionally, governance policies should account for licensing, attribution, and ethical compliance when LLM-generated code enters production.

Collectively, these recommendations establish a bridge between LLM-powered code generation/testing and real-world software engineering operations — addressing one of the major gaps identified in prior literature.

4. Discussion

The proposed integrated framework — combining taxonomy, evaluation guidelines, testing strategies, and infrastructure recommendations — represents a conceptual advance in how researchers and practitioners might systematically adopt LLMs in software engineering. However, several deeper interpretations, limitations, and future directions merit discussion.

Interpretation and Theoretical Implications

First, the taxonomy underscores that LLMs are not simply coding assistants but function as multi-faceted collaborators across software engineering: designers, testers, dev-ops assistants, and documentation writers. This multi-role capability challenges conventional role boundaries in software teams, potentially leading to redefined responsibilities, new workflows, and altered team dynamics. If LLMs reliably assume certain routine or boilerplate tasks (e.g., test generation, documentation), human developers may shift focus to higher-level design, architecture, or domain-specific logic. Such a shift has implications for educational curricula, skill development, and job roles in software engineering.

Second, the evaluation guidelines point to a necessary maturation of the field toward rigorous, multi-dimensional assessment. The historical tendency to evaluate code solely on execution correctness overlooks aspects such as maintainability, security, and long-term sustainability. By advocating for broader evaluation — including human review, long-term monitoring, and transparency — the framework aligns LLM research with established concerns in software engineering, such as technical debt, security risk management, and maintainability.

Third, the integrated testing strategy emphasizes that LLM-generated artifacts should not be treated differently just because they were machine-generated: they must undergo the same, if not more rigorous, validation than human-written code. This underscores an important theoretical point: the provenance of code (human vs. machine) should not dictate trust levels, but quality, reliability, and risk should. By integrating conventional and LLM-specific testing practices, we treat LLM outputs as first-class code artifacts deserving full lifecycle treatment.

Fourth, the infrastructure recommendations bring LLM-based development into the operational domain. Without scalable architecture, logging, CI/CD integration, and governance, LLM tools risk remaining experimental or siloed within small projects. The proposed deployment framework thus reflects a broader vision: LLM-powered development should not remain a novelty but become a stable, maintainable component of software engineering ecosystems.

5. Limitations

Despite the contributions, the article has inherent limitations, most stemming from its conceptual, non-empirical nature.

● Lack of Empirical Validation: The proposed framework is built through synthesis and logical reasoning, not through new experiments or large-scale deployments. As such, its practical effectiveness remains conjectural until validated by empirical studies.

● Dependence on Existing Literature Scope: The framework is constrained by the scope and limitations of the referenced works. For example, very few papers address long-term maintenance or real-world deployment, so sections on governance or operational scaling are necessarily speculative and informed by peripheral works (e.g., infrastructure papers) rather than solid empirical evidence.

● Rapidly Changing Landscape: The field of LLMs evolves extremely quickly. New architectures, deployment models, evaluation metrics, security vulnerabilities, and regulatory frameworks may emerge, potentially rendering parts of the framework obsolete.

● Human Factors Underexplored: While the framework acknowledges the need for human review and integration into team workflows, it does not deeply address social, organizational, or psychological factors — e.g., how developer trust evolves, how code ownership is perceived when parts are machine-generated, or how responsibility and accountability are managed.

● Ethical, Legal, and Licensing Issues: Although we mention governance and compliance, the framework does not deeply engage with complex issues such as licensing of training data, intellectual property ownership, and liability in case of bugs or security flaws. These are critical concerns but require dedicated legal and ethical scholarship.

Future Research Directions

Given these limitations, we outline several avenues for future research and development:

1. Empirical Implementation and Case Studies: Researchers should implement the proposed framework in real-world software development teams, across diverse domains (web, embedded systems, enterprise, open-source), to evaluate its practical viability, benefits, and limitations. Such case studies could measure metrics like developer productivity, code quality, bug incidence, time-to-release, and maintenance cost.

2. Benchmark Standardization and Open Datasets: There is a pressing need for community-wide benchmark suites for LLM-generated code—covering a variety of languages, domains, complexity levels, and real-world tasks. Such benchmarks should be open-source and collectively maintained, enabling better cross-study comparisons.

3. Governance, Licensing, and Ethical Frameworks: Dedicated work should be done to define policies for licensing, attribution, liability, and compliance when using LLM-generated code in commercial or open-source projects. Collaboration between legal scholars, ethicists, and software engineering researchers is vital.

4. Human-Centric Studies on Adoption and Trust: Empirical studies should investigate how human developers perceive and interact with LLM-generated code: Do they trust it? How often do they override or rewrite it? How does it affect code ownership and collaboration? What are the social dynamics involved?

5. Security and Robustness Analysis: Comprehensive security audits should be conducted on LLM-generated code, including dependency tracing, vulnerability scanning, and dynamic security testing. Research should also explore how adversarial inputs or malicious prompts could lead to insecure or buggy code.

6. Scaling and Infrastructure Tooling: Development of open-source platforms or tools that encapsulate the proposed deployment architecture would facilitate adoption. This includes container orchestration templates, CI/CD integrations, logging and monitoring dashboards, and compliance modules.

7. Policy and Standardization Initiatives: Professional bodies (e.g., IEEE, ISO) should consider extending existing software engineering standards (such as ISO/IEC/IEEE 24765:2017) to explicitly address AI-generated code, defining vocabulary, best practices, and quality standards.

6. Conclusion

The acceleration of research and practical efforts around Large Language Models has opened promising opportunities for their integration into software engineering workflows. However, fragmented investigations and varying methodologies have hindered consistent adoption and robust evaluation. This article synthesizes diverse strands of literature to propose a unified conceptual framework for integrating LLMs into software engineering — spanning taxonomy, evaluation guidelines, testing strategies, and deployment infrastructure.

By treating LLM-generated artifacts as first-class code, subject to the same rigorous testing, review, and governance as human-written code, the framework offers a pathway toward responsible, maintainable, and scalable adoption. While conceptual, the framework lays a foundation for future empirical validation, tool development, and standardization efforts.

Given the rapid pace of LLM development and growing industrial interest, adopting such a structured approach is not merely desirable — it may become essential. Only through systematic evaluation, disciplined testing, and responsible deployment can the promise of LLMs be responsibly realized in practical software engineering environments. We hope this framework stimulates further research, collaboration, and real-world implementations bridging the gap between LLM capabilities and software engineering best practices.

References

1. Zhao, W.X.; Zhou, K.; Li, J.; Tang, T.; Wang, X.; Hou, Y.; Min, Y.; Zhang, B.; Zhang, J.; Dong, Z.; et al. A Survey of Large Language Models. arXiv, 2024.
2. 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, 911–936.
3. Chen, L.; Guo, Q.; Jia, H.; Zeng, Z.; Wang, X.; Xu, Y.; Wu, J.; Wang, Y.; Gao, Q.; Wang, J.; et al. A Survey on Evaluating Large Language Models in Code Generation Tasks. arXiv, 2024.
4. Raiaan, M.A.K.; Mukta, M.d.S.H.; Fatema, K.; Fahad, N.M.; Sakib, S.; Mim, M.M.J.; Ahmad, J.; Ali, M.E.; Azam, S. A Review on Large Language

Models: Architectures, Applications, Taxonomies, Open Issues and Challenges. IEEE Access, 2024, 12, 26839–26874.

5. Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; Zhang, J.M. Large Language Models for Software Engineering: Survey and Open Problems. In Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Melbourne, Australia, 14–20 May 2023; pp. 31–53.
6. ISO/IEC/IEEE 24765:2017(E); ISO/IEC/IEEE International Standard — Systems and Software Engineering — Vocabulary. IEEE: New York, NY, USA, 2017.
7. Mayeda, M.; Andrews, A. Evaluating Software Testing Techniques: A Systematic Mapping Study. In Advances in Computers; Missouri University of Science and Technology: Rolla, MO, USA, 2021.
8. Lonetti, F.; Marchetti, E. Emerging Software Testing Technologies. In Advances in Computers; Elsevier: Amsterdam, The Netherlands, 2018, Volume 108, pp. 91–143.
9. Clark, A.G.; Walkinshaw, N.; Hierons, R.M. Test Case Generation for Agent-Based Models: A Systematic Literature Review. Information and Software Technology, 2021, 135, 106567.
10. Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; Wang, H. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv, 2024.
11. Chandra, R. Design and implementation of scalable test platforms for LLM deployments. Journal of Electrical Systems, 2025, 21(1s), 578–590.
12. Vasireddy, I.; Ramya, G.; Kandi, P. Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges. International Journal of Innovative Research in Engineering and Management, 2023, 10(6), 49–54.
13. Zhou, Y.; et al. Etbench: Characterizing Hybrid Vision Transformer Workloads Across Edge Devices. IEEE Transactions on Computers, 2025.
14. Borra, P. Comparison and analysis of leading cloud service providers (AWS, Azure and GCP). International Journal of Advanced Research in Engineering and Technology, 2024, 15, 266–278.
15. Pogiatzis, A.; Samakovitis, G. An Event-Driven Serverless ETL Pipeline on AWS. Applied Sciences, 2020, 11(1), 191.