COLOMBO SCIENTIFIC PUBLISHING

INTERNATIONAL JOURNAL OF COMPUTER SCIENCE & INFORMATION SYSTEM
E-ISSN: 2536-7919
P-ISSN: 2536-7900
https://doi.org/10.55640/ijcsis/Volume11Issue09-03 PAGE NO: 19-26

Secure and Trustworthy LLM-Based Code Generation for Enterprise Software Systems

Arjun Mehta
Department of Computer Science and Engineering,
National Institute of Software Technology, Bengaluru, India

ARTICLE INFO

Article history:

Submission: July 01, 2026
Accepted: August 28, 2026
Published: September 07, 2026
VOLUME: Vol.11 Issue 09 2026

Keywords:
Large Language Models, Code Generation, Software Security, Trustworthy AI, Enterprise Software, Automated Testing, Code Verification, Continuous Integration, Flaky Tests, Program Analysis

ABSTRACT

Large Language Models (LLMs) have introduced a new paradigm for software development by enabling developers to generate, transform, explain, and test source code through natural-language interaction. Their adoption in enterprise environments, however, introduces significant concerns regarding reliability, verification, security, maintainability, and confidence in generated artifacts. Enterprise software operates under stringent requirements for correctness, continuous integration, traceability, and operational stability, making uncontrolled reliance on automatically generated code problematic. This research-and-review paper develops a conceptual framework for secure and trustworthy LLM-based code generation by integrating LLM-assisted development with established principles of software testing, automated verification, static and dynamic program analysis, continuous integration, and flaky-test management. The methodology synthesizes the provided literature to identify how testing reliability, failure diagnosis, test prioritization, and large-scale continuous testing can serve as complementary safeguards for LLM-generated code. The analysis indicates that trustworthy code generation should not be treated as a generation-only problem; rather, it should be implemented as a closed-loop process consisting of generation, validation, testing, diagnosis, risk assessment, and controlled integration. Particular attention is given to flaky tests because nondeterministic validation can undermine confidence in otherwise correct code. The resulting framework emphasizes layered verification, automated quality gates, explainable validation evidence, and human oversight for high-risk enterprise components. The study contributes a research-oriented model for integrating LLM capabilities with mature software assurance mechanisms and identifies directions for future empirical evaluation.

INTRODUCTION

Large Language Models are increasingly being incorporated into software engineering workflows because they can generate source code from natural-language requirements, suggest implementations, repair defects, produce test cases, and assist developers during maintenance. In enterprise software development, these capabilities can potentially reduce development effort and accelerate delivery. Nevertheless, enterprise environments impose stronger requirements than isolated programming tasks. Generated code must satisfy functional requirements while also complying with organizational security policies, architectural constraints, testing standards, maintainability requirements, and deployment controls.

The central challenge is therefore not merely whether an LLM can generate syntactically valid code, but whether the resulting code can be trusted within a complex software lifecycle. Trustworthiness requires evidence that generated artifacts behave consistently, satisfy intended requirements, and remain stable under repeated execution. This concern is particularly important because conventional automated tests may themselves produce misleading signals. Research on flaky tests demonstrates that tests can fail intermittently without corresponding changes in the underlying software, complicating the interpretation of automated validation results (Luo et al., 2014; Parry et al., 2021).

The problem becomes more significant at enterprise scale. Continuous integration systems may execute thousands of tests repeatedly, and false alarms can consume developer attention while obscuring genuine defects. Studies of large-scale continuous testing and test-failure analysis demonstrate the importance of distinguishing meaningful failures from unreliable test outcomes (Memon, 2017; Jiang et al., 2017). Consequently, an LLM-based coding system requires more than a code-generation component; it requires an assurance architecture capable of evaluating generated artifacts throughout the software development lifecycle.

Recent work specifically emphasizes the need for secure and trustworthy LLM-assisted code generation in enterprise software development (Kongari et al., 2026). Building upon this perspective, the present study addresses the following research question: How can LLM-based code generation be integrated with software verification and testing mechanisms to improve security, reliability, and trustworthiness in enterprise software systems?

The objective is to develop a conceptual framework in which LLM-generated code is continuously subjected to verification, testing, failure diagnosis, and controlled integration. The study particularly focuses on the interaction between generated code and software quality-assurance mechanisms rather than treating LLM generation as an isolated artificial-intelligence task.

LITERATURE REVIEW

The literature supplied for this study provides a strong foundation for understanding the assurance mechanisms required around automated software generation. Although the references do not directly constitute a unified body of LLM-code-generation research, collectively they describe the testing, verification, diagnosis, and continuous-integration infrastructure needed to evaluate automatically produced software artifacts.

Luo et al. (2014) provide empirical evidence concerning flaky tests, establishing that test outcomes cannot always be interpreted as direct indicators of software correctness. This is particularly relevant to LLM-generated code because an automated generation pipeline may depend heavily on test execution for validation. If the testing layer is unreliable, confidence in the generated code can become distorted. Parry et al. (2021) further consolidate the field through a broad survey of flaky tests, demonstrating that flakiness represents a persistent software-engineering challenge rather than an isolated testing anomaly.

The developer-oriented investigation of flaky tests by Eck et al. (2019) adds an important human dimension. Developers must interpret and respond to unstable test outcomes, meaning that trustworthy automation requires mechanisms that communicate why validation has succeeded or failed. Similarly, Herzig et al. (2015) investigate false test alarms, while Herzig and Nagappan (2015) examine the detection of false alarms using association rules. These studies indicate that a trustworthy development pipeline should distinguish genuine defects from misleading validation signals.

The literature also demonstrates the importance of automated failure diagnosis. Jiang et al. (2017) investigate automatic cause analysis for test alarms, providing a conceptual basis for incorporating diagnostic mechanisms into LLM-assisted development. Ziftci and Reardon (2017) examine the identification of changes that induce test failures in continuous integration, illustrating how change-aware analysis can connect failures to specific modifications. Such approaches are relevant to LLM-generated code because a single generated change may modify multiple interacting components.

Enterprise-scale testing is another important dimension. Memon (2017) examines continuous testing at Google scale, while Micco (2017) discusses continuous integration testing at Google. Kowalczyk et al. (2020) investigate modeling and ranking flaky tests in an industrial environment, and Rehman and Rigby (2021) examine no-fault-found test failures in an Ericsson setting. These studies collectively demonstrate that software assurance at enterprise scale requires prioritization and systematic failure classification rather than indiscriminate test execution.

Lam et al. (2019) investigate root-cause analysis of flaky tests in a large industrial setting, while Lam et al. (2020) study the lifecycle of flaky tests. Leesatapornwongsa et al. (2022) address automated reproduction of concurrency-related flaky tests, illustrating the value of automated reproduction when nondeterministic behavior complicates diagnosis. Malm et al. (2020) investigate flakiness-mitigating delays, further demonstrating that the reliability of testing infrastructure must itself be actively managed.

Harman and O'Hearn (2018) position static and dynamic program analysis as important opportunities for software engineering. Their work supports a layered assurance model in which testing is complemented by analysis techniques capable of examining program properties from different perspectives. The provided testing and verification resources likewise reinforce the broader principle that verification must be incorporated into the software lifecycle rather than performed as an isolated final-stage activity.

The principal research gap is therefore architectural. The supplied literature provides extensive evidence for reliable testing, diagnosis, continuous integration, and program analysis, while enterprise LLM-assisted development requires these mechanisms to be coordinated around AI-generated artifacts. Kongari et al. (2026) explicitly situate trustworthy and secure LLM-assisted code generation within enterprise software development, motivating an integrated approach in which generation and verification operate as interconnected processes rather than independent activities.

METHODOLOGY

3.1 Research Design

This study adopts a conceptual research-and-review methodology based exclusively on the supplied references. The objective is not to evaluate a particular commercial LLM, but to synthesize established software-assurance concepts into a framework appropriate for LLM-based enterprise code generation.

The methodology consists of five analytical stages: requirement interpretation, code generation, multi-layer verification, test-result analysis, and controlled integration. Each stage addresses a different source of risk. The resulting architecture treats generated code as an artifact requiring evidence-based validation before it can enter an enterprise development branch.

3.2 LLM-Based Code Generation Layer

The first layer receives structured software requirements and generates candidate implementations. In a practical enterprise setting, the input can include functional requirements, architectural constraints, interface specifications, coding conventions, and testing expectations.

A critical methodological principle is that generated code should be considered candidate code, not automatically accepted production code. This distinction is essential because generation establishes a proposed implementation but does not independently establish correctness or security.

For example, an enterprise application may require an authentication service. An LLM can generate the controller, service layer, data-access code, and associated tests. The proposed implementation must subsequently pass organizational validation rules and automated testing before integration.

3.3 Static and Dynamic Verification

The second layer combines static and dynamic analysis. Static analysis examines source code without executing it, whereas dynamic analysis evaluates behavior during execution. The combination is theoretically valuable because different classes of defects may be observable through different analytical mechanisms.

Harman and O'Hearn (2018) emphasize the continuing importance of static and dynamic program analysis. In an LLM-assisted pipeline, static analysis can serve as an early quality gate before expensive integration

tests are executed. Generated code containing suspicious patterns can therefore be rejected or returned to the generation stage for correction.

Dynamic testing then evaluates functional behavior. However, because flaky tests can produce misleading outcomes, test results should be classified according to reliability rather than interpreted as binary truth signals.

3.4 Test Reliability and Flakiness Management

The third methodological component explicitly addresses flaky testing. A trustworthy LLM pipeline should maintain historical information about test stability and distinguish deterministic failures from intermittent failures.

The conceptual process can be represented as:

Generated Code → Static Analysis → Test Selection → Test Execution → Failure Classification → Root-Cause Analysis → Regeneration/Correction → Revalidation

Suppose an LLM generates a modification that causes an integration test to fail. If the test has historically been unstable, the system should avoid immediately concluding that the generated code is defective. Instead, the failure can be reproduced and correlated with historical test behavior. The findings of Luo et al. (2014), Lam et al. (2019), and Leesatapornwongsa et al. (2022) support the importance of this distinction.

Test ranking can further reduce validation cost. Evidence from industrial settings indicates that test prioritization can help focus developer attention on the most informative failures (Kowalczyk et al., 2020; Rehman and Rigby, 2021).

3.5 Failure Diagnosis and Explainable Validation

The fourth layer is failure diagnosis. Instead of reporting only that generated code failed, the system should provide evidence regarding the failure's likely origin.

A useful diagnostic record can include the generated change, affected components, failed tests, historical stability of those tests, execution context, and suspected root cause. Jiang et al. (2017) and Ziftci and Reardon (2017) provide conceptual support for automated cause analysis and change-failure association.

This layer is essential for trust because developers need actionable evidence. A system that states “generation failed” provides limited assistance, whereas a system that identifies a probable dependency regression and links it to a specific generated modification can support informed human review.

3.6 Continuous Integration and Enterprise Governance

The fifth layer integrates the assurance process with continuous integration. Enterprise software requires repeated validation because code changes interact with rapidly evolving repositories.

The studies of Memon (2017) and Micco (2017) illustrate the importance of continuous testing at large organizational scale. Accordingly, the proposed architecture places LLM-generated changes inside existing integration controls rather than allowing the AI system to bypass them.

Kongari et al. (2026) further reinforces the enterprise perspective by framing trustworthy LLM-assisted development around security and reliability. Thus, the proposed methodology treats governance as an operational control: generated code can progress from isolated validation to integration only when defined quality gates are satisfied.

RESULTS

The synthesis produces five principal findings concerning trustworthy LLM-based enterprise code generation.

First, generation and verification must be architecturally separated but operationally connected. LLMs are effective as code-producing mechanisms, but generated output should remain provisional until it has passed automated and human-defined validation gates. This approach reduces the risk of treating fluent or syntactically correct code as inherently reliable.

Second, testing reliability is a prerequisite for trustworthy AI-assisted development. The literature consistently demonstrates that flaky tests can generate misleading signals. Luo et al. (2014) empirically examine flaky tests, while Parry et al. (2021) demonstrate their broader significance. Therefore, an enterprise LLM pipeline that ignores test stability can incorrectly reject valid generated code or accept defective code based on unreliable evidence.

Third, failure diagnosis substantially increases the usefulness of automated validation. Jiang et al. (2017) and Ziftci and Reardon (2017) demonstrate the value of connecting test alarms with probable causes or responsible changes. Applied to LLM-generated modifications, this principle enables the system to move from simple pass/fail validation toward evidence-based correction.

Fourth, enterprise-scale validation requires prioritization. Continuous testing environments can generate large numbers of results, making indiscriminate human inspection inefficient. Industrial studies of flaky-test ranking and no-fault-found failures indicate that prioritization mechanisms can help allocate attention to higher-value signals (Kowalczyk et al., 2020; Rehman and Rigby, 2021). In an LLM-assisted environment, this can be used to determine which generated changes require immediate human review.

Fifth, trustworthiness emerges from layered assurance rather than a single verification technique. Static analysis, dynamic testing, test-history analysis, failure reproduction, root-cause analysis, and continuous integration address different failure modes. Their combination creates stronger evidence than any individual mechanism.

The resulting framework can consequently be interpreted as a closed-loop assurance architecture. When generated code fails a reliable validation gate, diagnostic evidence is returned to the correction or regeneration stage. When failure originates from an unstable test, the pipeline can prioritize test investigation rather than automatically rejecting the code. When all relevant quality gates pass, the change can proceed to controlled integration.

Overall, the findings support the proposition that secure and trustworthy LLM-assisted development depends less on unconditional confidence in the model and more on the engineering controls surrounding the model. This finding aligns with the enterprise-oriented perspective of Kongari et al. (2026), in which trustworthiness is treated as a property of the complete development process rather than solely of the generation model.

DISCUSSION

The findings have important theoretical implications for software engineering and trustworthy AI. Traditional code-generation paradigms generally emphasize productivity and syntactic correctness, whereas enterprise development requires a broader notion of software quality. The proposed framework conceptualizes trust as an emergent property of interacting components: generated code, verification mechanisms, testing infrastructure, diagnostic processes, and governance controls.

One significant implication concerns the role of testing. It is tempting to use automated tests as a definitive oracle for LLM-generated code, but the literature on flaky tests challenges this assumption. A failing test does not necessarily indicate a defect in the latest code change, and a passing test does not establish complete correctness. Therefore, the proposed framework treats test outcomes as evidence whose reliability must itself be assessed.

This creates an important trade-off. More extensive verification can increase confidence but also increases computational and organizational cost. Enterprise systems may contain thousands of tests, dependencies, and services. Running every possible validation procedure for every generated modification may undermine the productivity advantages of LLM assistance. Test prioritization and risk-based validation therefore become necessary. The industrial evidence concerning test ranking and continuous testing supports such selective strategies.

Another trade-off concerns automation versus human oversight. Fully autonomous code integration may maximize development speed but introduces greater consequences when validation mechanisms fail to detect subtle defects. Conversely, excessive human review can reduce the productivity benefits of LLMs. A balanced architecture should therefore reserve mandatory human approval for high-risk changes, security-sensitive components, architectural modifications, and ambiguous validation results while allowing lower-risk changes to proceed through automated gates.

The framework also has implications for explainability. Trust cannot be established solely through a numerical confidence score generated by an AI model. Developers need traceable evidence concerning which tests were executed, which analyses were performed, whether those tests are stable, and why a change was accepted or rejected. The research on failure diagnosis and test alarms suggests that explanatory evidence can make automated quality assurance more actionable.

A further limitation is that the supplied literature primarily addresses software testing and program-analysis challenges rather than comprehensive LLM-specific security evaluation. Consequently, the framework should be understood as a software-assurance architecture for LLM-generated code rather than a complete solution to every AI-specific threat. Empirical evaluation with multiple LLMs, programming languages, enterprise repositories, and security-sensitive applications is required before the framework can be considered universally validated.

Finally, the framework reinforces the central argument that trustworthy LLM-assisted software development should be engineered as a lifecycle process. Kongari et al. (2026) similarly emphasize the security and trustworthiness requirements associated with enterprise LLM-assisted development. The present synthesis extends that perspective by explicitly connecting those requirements with established evidence from continuous integration, automated testing, failure diagnosis, and flaky-test research.

CONCLUSION

This study presented a conceptual framework for secure and trustworthy LLM-based code generation in enterprise software systems. The central conclusion is that trustworthy code generation cannot be achieved through model generation capability alone. Generated source code should be treated as a candidate artifact that must pass layered verification, reliable testing, failure diagnosis, and enterprise integration controls.

The literature demonstrates that flaky tests, false alarms, large-scale continuous integration, and difficult failure diagnosis can undermine conventional software assurance. These challenges become particularly important when LLMs increase the volume and velocity of generated code. The proposed framework therefore integrates static analysis, dynamic testing, test-reliability assessment, failure reproduction, root-cause analysis, test prioritization, and controlled continuous integration into a closed-loop development process.

The principal contribution is an integrated assurance perspective in which LLM generation is embedded within established software-engineering controls. Such an approach can improve developer confidence while retaining the productivity benefits of AI-assisted development. It also establishes a basis for explainable validation by requiring generated artifacts to be accompanied by evidence about their testing and analysis outcomes.

Future research should empirically evaluate the framework across enterprise-scale repositories and compare different LLM-assisted development strategies. Further work should investigate risk-based test selection, automated security-policy enforcement, explainable generation feedback, and quantitative trust

metrics. Longitudinal studies could also examine whether integrating test-history and failure-diagnosis information reduces false acceptance and false rejection of AI-generated code. Ultimately, secure enterprise adoption of LLM code generation will depend on treating AI not as a replacement for software assurance, but as one component within a rigorously governed engineering lifecycle.

REFERENCES

1. "Facebook testing and verification request for proposals," Accessed: Feb. 4, 2023. [Online]. Available: https://research.fb.com/programs/research-awards/proposals/facebook-testing-and-verification-request-for-proposals-2019

2. "Test verification," Accessed: Feb. 4, 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification

3. "TotT: Avoiding flakey tests," Google. Accessed: Feb. 4, 2023. [Online]. Available: http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html

4. M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in Proc. 27th Joint Meeting Found. Softw. Eng., 2019, pp. 830–840.

5. M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in Proc. 18th Int. Work. Conf. Source Code Anal. Manipulation (SCAM), 2018, pp. 1–23.

6. K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in Proc. 37th Int. Conf. Softw. Eng. (ICSE), 2015, pp. 39–48.

7. K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in Proc. 37th Int. Conf. Softw. Eng. (ICSE), 2015, pp. 483–493.

8. H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing," in Proc. 39th Int. Conf. Softw. Eng. (ICSE), 2017, pp. 712–723.

9. E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in Proc. 42nd Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE SEIP), 2020, pp. 110–119.

10. W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in Proc. Int. Symp. Softw. Testing Anal. (ISSTA), 2019, pp. 101–111.

11. W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in Proc. 42nd Int. Conf. Softw. Eng. (ICSE), 2020, pp. 1471–1482.

12. T. Leesatapornwongsa, X. Ren, and S. Nath, "FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests," in Proc. 30th Joint Meeting Found. Softw. Eng. (ESEC/FSE), 2022, pp. 1509–1520.

13. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in Proc. ACM SIGSOFT 22nd Symp. Found. Softw. Eng., 2014, pp. 643–653.

14. J. Malm, A. Causevic, B. Lisper, and S. Eldh, "Automated analysis of flakiness-mitigating delays," in Proc. 1st Int. Conf. Automat. Softw. Test (AST), 2020, pp. 81–84.

15. A. Memon, "Taming Google-scale continuous testing," in Proc. 39th Int. Conf. Softw. Eng., Softw. Eng. Pract. Track (ICSE SEIP), 2017, pp. 233–242.

16. J. Micco, "The state of continuous integration testing at Google," in Proc. 10th Int. Conf. Softw. Testing, Verification Validation Keynote (ICST), 2017.

17. O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," Trans. Softw. Eng. Methodol., vol. 31, no. 1, pp. 1–74, 2021.

18. M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds," in Proc. 26th Joint Meeting Found. Softw. Eng. (ESEC/FSE), 2018, pp. 857–862.

19. M. H. U. Rehman and P. C. Rigby, "Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson," in Proc. 29th Joint Meeting Found. Softw. Eng., Ind. Track (ESEC/FSE), 2021.

20. C. Ziftci and J. Reardon, "Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale," in Proc. 39th Int. Conf. Softw. Eng. (ICSE), 2017, pp. 113–122.

21. Kongari, S. S. R., Kumar, S. K.., & Kumar, A. (2026). Trustworthy and Secure LLM-Assisted Code Generation for Enterprise Software Development. International Journal of Data Science and Machine Learning, 6(01), 222-237. https://doi.org/10.55640/ijdsml-06-01-04