FECSIT, (2026)

Frontiers in Emerging Computer Science and Information Technology Technologies

Volume 03, Issue 09, September 2026,
Publish Date: 07-09-2026
DOI: https:/doi.org/10.64917/fecsit/Volume03Issue09-02 **PageNo.07-13**

A Robust Framework for Reliable and Secure Large Language Model-Assisted Software Development

Amirul Rahman
Faculty of Computing and Software Engineering, Malaysian Institute of Digital Technology, Kuala Lumpur, Malaysia

Nur Aisyah Hassan
Faculty of Information Technology, Malaysian Institute of Digital Technology, Kuala Lumpur, Malaysia

ABSTRACT

Large Language Models (LLMs) are increasingly being incorporated into software-development workflows for code generation, test creation, debugging, refactoring, documentation, and maintenance. Although LLM-assisted development can improve developer productivity, generated code cannot automatically be regarded as reliable or secure. In particular, nondeterministic behavior in software tests, unreliable validation procedures, inadequate historical testing information, and insufficient root-cause analysis can weaken confidence in AI-generated software. This paper proposes a robust framework for reliable and secure LLM-assisted software development by integrating LLM-based code generation with systematic validation, flaky-test detection, historical test analysis, static prediction, and automated root-cause assessment. The framework synthesizes findings from existing research on flaky-test detection and prediction and extends their implications to AI-assisted development environments. The methodology defines a multi-stage pipeline covering requirement interpretation, constrained code generation, static analysis, test generation, flakiness assessment, reliability-oriented test prioritization, security validation, and human-controlled release decisions. The analysis indicates that reliability should be treated as a continuous property rather than a final-stage testing outcome. The proposed framework also highlights the importance of separating genuine software defects from nondeterministic test failures, since unreliable tests can distort the evaluation of generated code. The paper contributes a conceptual architecture for combining trustworthy LLM-assisted development with established software-quality mechanisms and identifies directions for empirical validation, adaptive risk scoring, and enterprise-scale deployment.

KEYWORDS: Large Language Models, Secure Code Generation, Software Development, Flaky Tests, Software Reliability, Test Prediction, Automated Testing, AI-Assisted Programming, Root Cause Analysis, Software Security

INTRODUCTION

Large Language Models have changed the operational model of software development by enabling developers to generate source code, test cases, documentation, debugging suggestions, and implementation alternatives through natural-language interaction. This development paradigm offers substantial potential for reducing repetitive programming effort and accelerating development cycles. However, the ability of an LLM to generate syntactically valid or functionally plausible code does not guarantee that the resulting software is reliable, maintainable, or secure. Trust therefore depends not only on the generation capability of the model but also on the mechanisms used to evaluate, validate, and control generated artifacts.A fundamental challenge is that software quality assessment itself may be unreliable. Flaky tests can produce inconsistent outcomes even when the underlying implementation remains unchanged. Research has demonstrated that flaky tests constitute a significant challenge for automated testing because their nondeterministic behavior can obscure the distinction between genuine failures and environmental or execution-related variability (Luo et al., 2014). In an LLM-assisted environment, this problem becomes particularly important because generated code may be evaluated through automatically produced tests at a much higher frequency than in conventional development.

Existing research has investigated several approaches for identifying and predicting flaky tests. DeFlaker introduced automated detection mechanisms for identifying tests whose failures are not necessarily associated with relevant code changes (Bell et al., 2018). Later approaches

investigated static prediction, historical test information, source-code vocabulary, machine-learning models, and language-model-based prediction (Verdecchia et al., 2021; Fallahzadeh and Rigby, 2022; Fatima et al., 2023). These studies collectively suggest that reliable software validation requires more than simple pass/fail testing.

The relevance of this observation extends directly to trustworthy LLM-assisted development. A code-generation system should not merely generate an implementation and execute a test suite. It should establish whether the validation evidence itself is dependable, whether repeated failures represent genuine defects, and whether generated code satisfies predefined security and reliability constraints. Recent work specifically addressing trustworthy and secure LLM-assisted code generation for enterprise software development reinforces the importance of treating security and trust as integral components of the development lifecycle (Kongari et al., 2026).

The objective of this paper is therefore to develop a conceptual framework for robust, reliable, and secure LLM-assisted software development. The framework integrates LLM generation with reliability-oriented testing mechanisms derived from the provided literature. Its scope covers code-generation workflows in which automatically generated software must undergo systematic validation before being accepted into an enterprise development pipeline.

The principal contribution is a layered framework in which generation, verification, test-reliability analysis, security assessment, and human oversight operate as complementary controls. Instead of assuming that successful test execution proves correctness, the framework treats test reliability as a prerequisite for trustworthy evaluation.

LITERATURE REVIEW

2.1 Flaky Tests as a Threat to Software Reliability

Luo et al. conducted an empirical analysis of flaky tests and established the importance of understanding failures that do not consistently correspond to deterministic defects (Luo et al., 2014). Their work provides a foundational perspective for the proposed framework: a testing pipeline cannot be considered reliable merely because it contains a large number of tests. The behavioral stability of those tests must also be considered.

The consequences of flaky tests extend beyond isolated testing inconvenience. Lam et al. presented a large-scale longitudinal study showing that flaky tests can persist and evolve within software projects (Lam et al., 2020). This longitudinal perspective is especially relevant to LLM-assisted development because AI-generated code may rapidly enter iterative development cycles. If unstable tests accumulate during these cycles, developers may increasingly encounter noisy validation signals.

Habchi et al. further examined the sources, impacts, and mitigation strategies associated with flaky tests, demonstrating that flakiness is a multidimensional software-quality problem rather than a single technical defect category (Habchi et al., 2022). This supports the framework's use of multiple validation mechanisms instead of relying on a single detection algorithm.

2.2 Automated Detection and Prediction

Bell et al. introduced DeFlaker as an automated approach for detecting flaky tests (Bell et al., 2018). Its significance for an AI-assisted development architecture lies in demonstrating how automated mechanisms can distinguish potentially unreliable test outcomes from conventional failures.

Alshammari et al. proposed FlakeFlagger, which focuses on predicting flakiness without requiring tests to be rerun (Alshammari et al., 2021). This characteristic is important for enterprise environments where repeated execution of large test suites may introduce considerable computational cost. A predictive reliability layer can prioritize suspicious tests before expensive validation operations are performed.

Ahmad et al. investigated a multi-factor approach for flaky-test detection and automated root-cause analysis (Ahmad et al., 2021). The multi-factor perspective is consistent with the proposed framework because test reliability should not be inferred from one signal. Source-code changes, historical behavior, execution patterns, and contextual characteristics can collectively contribute to a more robust assessment.

2.3 Historical and Static Evidence

Fallahzadeh and Rigby studied the impact of flaky tests on historical test prioritization in Chrome (Fallahzadeh and Rigby, 2022). Their findings highlight an important dependency: historical testing data can be valuable for prioritization, but unreliable historical outcomes can also affect subsequent decision-making.

Pontillo et al. examined static test-flakiness prediction as a feasibility problem (Pontillo et al., 2021), while Verdecchia et al. investigated fast static prediction of test flakiness (Verdecchia et al., 2021). These approaches indicate that reliability analysis can be integrated earlier into the testing workflow rather than being restricted to repeated execution.

Haben et al. examined whether code vocabulary could be used to predict flaky tests and subsequently conducted a replication study concerning the usability of this information (Haben et al., 2021). The replication

perspective is particularly valuable because predictive features that appear effective in one context should not automatically be considered universally reliable.

2.4 Machine Learning and Language-Model-Based Prediction

Gruber et al. investigated practical flaky-test prediction using common code-evolution and test-history data (Gruber et al., 2023). Their work reinforces the usefulness of combining development-history signals with test characteristics.

Fatima et al. advanced this direction through Flakify, a black-box language-model-based predictor for flaky tests (Fatima et al., 2023). This work is directly relevant to the proposed architecture because it demonstrates the applicability of language-model techniques to software-test reliability prediction. Importantly, using a language model for prediction does not eliminate the need for validation; instead, it introduces another predictive component whose outputs should themselves be interpreted as probabilistic evidence.

Qin's Peeler approach investigates learning to predict flakiness without executing tests (Qin, 2022). This supports the concept of shifting part of reliability assessment toward pre-execution analysis.

Cordy et al. examined the impact of test flakiness in controlled laboratory settings, including mutation testing and program repair (Cordy et al., 2019). Their work illustrates how unreliable tests can affect downstream software-engineering activities, reinforcing the need to account for test reliability before automated repair or code modification decisions are made.

Pinto et al. studied the vocabulary associated with flaky tests (Pinto et al., 2020), providing additional evidence that linguistic and source-code characteristics can contribute to flakiness analysis.

2.5 Research Gap and Theoretical Positioning

The provided literature contains substantial work on flaky-test detection, prediction, prioritization, and analysis. However, these studies primarily address conventional software-development and testing environments. A broader framework is required for settings where an LLM actively participates in code creation and modification.

The research gap is therefore not simply the absence of another flaky-test predictor. Rather, the gap concerns the integration of test reliability into a trustworthy LLM-assisted development lifecycle. An LLM-generated implementation can be evaluated incorrectly if the test suite contains unstable tests. Conversely, a genuine defect may be overlooked if repeated failures are incorrectly classified as flaky behavior.

The proposed framework addresses this gap by positioning flaky-test analysis as a reliability control layer within LLM-assisted software engineering. This conceptual positioning is consistent with the broader trustworthy and secure code-generation perspective proposed by Kongari et al. (2026).

METHODOLOGY

3.1 Proposed Framework Architecture

The proposed framework consists of seven interconnected stages:

Requirement and Risk Analysis → Controlled LLM Code Generation → Static and Security Validation → Test Generation and Execution → Flakiness Detection → Reliability-Aware Decision Engine → Human-Governed Deployment

The architecture is designed around a central principle: generated code should not proceed directly from LLM output to production. Each transition should generate evidence that can be evaluated by the next layer.

The first stage converts natural-language requirements into structured development constraints. These constraints include functional requirements, architectural boundaries, security policies, dependency restrictions, and testing expectations. Explicit constraints reduce ambiguity in the generation process and provide measurable criteria against which generated code can later be evaluated.

The second stage performs constrained code generation. Instead of accepting a single generated solution, the system may produce one or more candidate implementations. Each candidate is associated with its generation context, requested functionality, model configuration, and validation history. This metadata becomes important for traceability.

The third stage performs static and security-oriented validation. Static analysis examines syntax, type consistency, dependency relationships, unreachable logic, suspicious patterns, and other detectable characteristics. Security validation examines whether generated code violates predefined enterprise security policies. The framework does not assume that an LLM can reliably self-validate its own output; independent controls are therefore required.

3.2 Test Generation and Execution

Once a candidate implementation passes initial checks, functional tests are generated or selected. Testing should combine newly generated tests with existing regression tests where available. The objective is to evaluate both

newly introduced functionality and potential regression effects.

However, a test failure cannot immediately be interpreted as evidence of defective generated code. The framework therefore records execution history and contextual metadata. Variables such as test frequency, previous outcomes, code changes, and execution consistency contribute to the reliability assessment.

This approach follows the broader finding that flaky-test behavior can arise from multiple sources and can distort conventional software-testing processes (Habchi et al., 2022).

3.3 Flakiness Detection Layer

The flakiness layer assigns a reliability assessment to each test. Conceptually, a test reliability score can be represented as:

$R_t = 1 - P(F_t \mid H_t, C_t, E_t)$

where $P(F_t)$ represents the estimated probability that a test failure is caused by flaky behavior, H_t represents historical test behavior, C_t represents code-change information, and E_t represents execution-related evidence.

This formulation does not imply that the probability can be perfectly estimated. Instead, it provides a decision-oriented representation of uncertainty.

Static predictors such as those explored by Pontillo et al. and Verdecchia et al. can provide early indicators, while historical information and execution behavior can provide additional evidence (Pontillo et al., 2021; Verdecchia et al., 2021). Machine-learning and language-model-based predictors provide further opportunities for classification, as demonstrated by Gruber et al. and Fatima et al. (Gruber et al., 2023; Fatima et al., 2023).

3.4 Root-Cause Analysis

Detection alone is insufficient. When a failure is identified, the framework attempts to classify its likely cause into categories such as implementation defect, test defect, environmental instability, dependency instability, timing-related behavior, or suspected nondeterministic execution.

The multi-factor root-cause perspective investigated by Ahmad et al. provides a conceptual basis for this layer (Ahmad et al., 2021). A confidence score can be associated with each classification. Low-confidence classifications should trigger human review rather than automatic remediation.

This is particularly important for LLM-assisted development because an LLM might otherwise respond to a flaky failure by modifying correct code unnecessarily. Such behavior could introduce new defects while attempting to eliminate a failure that was never caused by the implementation.

3.5 Reliability-Aware Test Prioritization

The framework uses reliability information when prioritizing tests. Tests with high failure relevance and high reliability should generally receive greater priority than tests with repeated evidence of nondeterministic behavior.

Historical test prioritization can be affected by flaky tests, as demonstrated in the study of Chrome's testing environment by Fallahzadeh and Rigby (2022). Therefore, test history should not be treated as inherently trustworthy. Instead, historical evidence should be weighted according to its reliability.

A conceptual priority score can be defined as:

$P_i = D_i \times R_i \times C_i$

where D_i represents defect-detection relevance, R_i represents test reliability, and C_i represents the relevance of the test to recently changed code.

This mechanism allows the framework to balance testing value against reliability.

3.6 Security Control Layer

Reliability and security are related but distinct dimensions. A program may consistently pass functional tests while still containing vulnerabilities. Therefore, security validation must operate independently of test reliability assessment.

The security layer evaluates generated code against organizational policies, dependency restrictions, access-control requirements, data-handling rules, input-validation requirements, and other predefined security constraints. Security violations should result in rejection or mandatory review rather than being compensated for by strong functional test performance.

The central trust principle is therefore:

Trust = Functional Evidence + Test Reliability + Security Compliance + Traceability + Human Oversight

This formulation treats trust as a composite property rather than a direct consequence of model confidence. Such an approach is aligned with the need for trustworthy and secure LLM-assisted code-generation processes identified by Kongari et al. (2026).

3.7 Human-Governed Decision Engine

The final stage combines validation evidence into an acceptance decision. Three broad outcomes are possible: automatic acceptance for low-risk changes, conditional

acceptance requiring additional verification, or rejection requiring modification.

Human oversight remains necessary for high-risk software components, ambiguous requirements, low-confidence predictions, security exceptions, and conflicting evidence. The framework therefore follows a human-in-the-loop rather than a fully autonomous model.

RESULTS

The conceptual analysis produces five principal findings.

First, LLM code-generation reliability depends on the reliability of the validation pipeline. A generated implementation cannot be evaluated meaningfully when its principal tests are unstable. Research on flaky tests consistently demonstrates that nondeterministic behavior can complicate interpretation of test outcomes (Luo et al., 2014; Lam et al., 2020).

Second, flakiness detection should be integrated before final software-quality decisions. Approaches based on static characteristics, historical information, and predictive models demonstrate that flakiness can be assessed using evidence beyond repeated execution (Pontillo et al., 2021; Qin, 2022; Verdecchia et al., 2021). This creates an opportunity to reduce unnecessary test reruns and improve validation efficiency.

Third, multiple evidence sources provide a stronger reliability foundation than isolated indicators. The multi-factor approach investigated by Ahmad et al. and the combination of code evolution and test-history information examined by Gruber et al. support a composite assessment strategy (Ahmad et al., 2021; Gruber et al., 2023). The proposed framework consequently avoids treating any single predictor as authoritative.

Fourth, test reliability must influence automated development decisions. If a generated code change fails primarily because of a historically unstable test, automatic code repair may be inappropriate. FlakiMe's investigation of flakiness effects on mutation testing and program repair illustrates why downstream automation can be influenced by unreliable tests (Cordy et al., 2019).

Fifth, security cannot be inferred from functional correctness. A trustworthy LLM-assisted development pipeline therefore requires independent security controls, traceability, and human review. The framework's reliability layer should complement rather than replace security governance, consistent with the broader trustworthy-code-generation perspective of Kongari et al. (2026).

Overall, the findings indicate that robustness emerges from the interaction of generation controls, validation reliability, predictive testing, security assessment, and human governance rather than from the LLM alone.

DISCUSSION

The proposed framework changes the interpretation of reliability in AI-assisted software development. Conventional development workflows frequently treat testing as the principal evidence of implementation quality. The literature reviewed here demonstrates that this assumption is incomplete because test outcomes themselves can be unreliable. In an LLM-assisted environment, the issue becomes more consequential because automated code production increases the number of implementation and validation cycles.

A major theoretical implication is that validation reliability should be modeled as a first-class software-quality property. Rather than representing quality as a simple function of successful test execution, the framework separates implementation correctness from confidence in the evidence used to establish correctness. This distinction is particularly important when generated code is evaluated through automatically generated tests.

The framework also creates a trade-off between validation depth and development speed. Static flakiness prediction can reduce unnecessary execution, as explored by approaches such as FlakeFlagger and Peeler (Alshammari et al., 2021; Qin, 2022). However, prediction introduces uncertainty. A false positive may cause a useful test to receive insufficient attention, whereas a false negative may allow an unstable test to influence an important software decision. Consequently, predictive models should support prioritization rather than function as unquestionable decision authorities.

Another issue concerns generalization. Haben et al.'s replication study demonstrates the importance of evaluating whether predictive signals remain useful across different development contexts (Haben et al., 2021). Likewise, vocabulary-based or language-model-based indicators may perform differently across programming languages, repositories, organizations, and test architectures. A robust enterprise framework should therefore monitor prediction quality continuously rather than assuming permanent model validity.

The literature also reveals an important distinction between detection and explanation. Detecting a potentially flaky test does not establish why it is flaky. Root-cause analysis is consequently essential for trustworthy automated remediation. The multi-factor approach of Ahmad et al. supports this principle, while the qualitative findings of Habchi et al. indicate that flakiness can arise from diverse causes (Ahmad et al., 2021; Habchi et al., 2022).

For LLM-assisted development, this distinction has direct practical implications. Suppose an AI-generated authentication component fails intermittently because a test depends on execution timing. Automatically modifying the authentication logic may reduce the observed failure temporarily but could also weaken security or introduce functional defects. A reliability-aware framework would first investigate the test's stability and causal evidence before permitting code modification.

The framework nevertheless has limitations. It is conceptual and has not been empirically validated against a large corpus of LLM-generated enterprise software. Its proposed reliability score is therefore a design abstraction rather than a demonstrated predictive metric. Furthermore, security validation is represented as a dedicated architectural layer without a domain-specific vulnerability benchmark. Future empirical studies should evaluate the framework using real repositories, controlled LLM-generation experiments, diverse test suites, and security-focused benchmarks.

A further limitation is the potential computational cost of maintaining historical evidence, executing multiple validation layers, and performing root-cause analysis. Enterprise adoption therefore requires adaptive policies that allocate verification effort according to software risk. High-risk components should receive stronger validation, while low-risk changes may use lighter-weight controls.

Despite these limitations, the framework provides a coherent theoretical bridge between research on flaky-test reliability and emerging trustworthy LLM-assisted software engineering. The central argument is that reliable AI-generated code requires reliable evidence, and reliable evidence requires explicit analysis of the testing mechanisms used to produce it.

CONCLUSION

This paper proposed a robust framework for reliable and secure Large Language Model-assisted software development. The framework addresses a central limitation of AI-assisted programming: generated code cannot be considered trustworthy solely because it appears plausible or passes an initial test suite.

By synthesizing research on flaky-test detection, prediction, historical analysis, static assessment, root-cause analysis, and language-model-based prediction, the paper positioned test reliability as an explicit control layer within the LLM-assisted development lifecycle. The proposed architecture integrates requirement analysis, constrained code generation, static validation, security assessment, test generation, flakiness prediction, root-cause analysis, reliability-aware test prioritization, and human-governed deployment.

The principal contribution is a conceptual shift from code generation followed by testing toward evidence-driven generation and validation. In this model, test results are treated as evidence whose reliability must itself be assessed. This distinction can reduce inappropriate automated remediation, improve test prioritization, and strengthen confidence in software-development decisions.

The framework also demonstrates that reliability and security should remain complementary dimensions. Reliable testing does not guarantee security, while security validation does not guarantee functional correctness. A trustworthy development process therefore requires both, together with traceability and appropriate human oversight. This perspective is consistent with the broader objective of trustworthy and secure LLM-assisted code generation (Kongari et al., 2026).

Future research should empirically evaluate the framework using large-scale enterprise repositories and diverse LLM-generated software. Further investigation should examine adaptive reliability scoring, cross-project generalization of flakiness predictors, security-aware code-generation policies, automated causal analysis, and risk-based allocation of validation resources. Such studies can determine whether the proposed architecture can provide measurable improvements in software reliability, development efficiency, and security while preserving the productivity benefits of LLM-assisted programming.

REFERENCES

1. A. Ahmad, F. G. Oliveira Neto, Z. Shi, K. Sandahl, and O. Leifler, “A multi-factor approach for flaky test detection and automated root cause analysis,” in Proc. 28th Asia-Pacific Softw. Eng. Conf. (APSEC), Piscataway, NJ, USA : IEEE Press, 2021, pp. 338–348.
2. A. Alshammari, C. Morris, M. Hilton, and J. Bell, “FlakeFlagger: Predicting flakiness without rerunning tests,” in Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE), Piscataway, NJ, USA : IEEE Press, 2021, pp. 1572–1584.
3. J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically detecting flaky tests,” in Proc. 40th Int. Conf. Softw. Eng., 2018, pp. 433–444.
4. M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, “FlakiMe: Laboratory-controlled test flakiness impact assessment. A case study on mutation testing and program repair,” 2019, arXiv:1912.03197.
5. E. Fallahzadeh and P. C. Rigby, “The impact of flaky tests on historical test prioritization on chrome,” in Proc.

44th Int. Conf. Softw. Eng.: Softw. Eng. Pract., 2022, pp. 273–282.

6. S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," IEEE Trans. Softw. Eng., vol. 49, no. 4, pp. 1912–1927, Apr. 2023.
7. M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser, "Practical flaky test prediction using common code evolution and test history data," in Proc. IEEE Conf. Softw. Testing, Verification Validation (ICST), Piscataway, NJ, USA : IEEE Press, 2023, pp. 210–221.
8. S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. L. Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in Proc. IEEE Conf. Softw. Testing, Verification Validation (ICST), Piscataway, NJ, USA : IEEE Press, 2022, pp. 244–255.
9. G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR), Piscataway, NJ, USA : IEEE Press, 2021, pp. 219–229.
10. B. Harry, "How we approach testing VSTS to enable continuous delivery," 2019. [Online]. Available: https://devblogs.microsoft.com/bharry/testing-in-a-cloud-delivery-cadence/
11. W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," Proc. ACM Program. Lang., vol. 4, no. OOPSLA, pp. 1–29, 2020.
12. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng., 2014, pp. 643–653.
13. G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests? " in Proc. 17th Int. Conf. Mining Softw. Repositories, 2020, pp. 492–502.
14. V. Pontillo, F. Palomba, and F. Ferrucci, "Toward static test flakiness prediction: A feasibility study," in Proc. 5th Int. Workshop Mach. Learn. Techn. Softw. Qual. Evol., 2021, pp. 19–24.
15. Y. Qin, "Peeler: Learning to effectively predict flakiness without running tests," in Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME), Piscataway, NJ, USA : IEEE Press, 2022, pp. 257–268.
16. R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," IEEE Access, vol. 9, pp. 76119–76134, 2021.
17. 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