IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

Semantic-Aware LLM-Based Test Generation: A Closed-Loop Framework for Requirement Alignment and Fault Detection

Sooraj Jacob[1], Rajeew Vishvakarma[2]

Software Engineer, Florida (U.S.A.) IntraEdge[1]

Project Manager, Florida (U.S.A.), Infosys[2]

Abstract: Large language models (LLMs) increasingly generate software tests from source codes, natural language requirements, and API specifications. While LLMs can produce readable test cases with plausible assertions, the generated tests may fail to validate requirement intent, omit boundary conditions, duplicate scenarios, or provide weak fault detection. This study proposes a semantic-aware closed-loop framework for LLM-based test generation. The framework combines requirement ingestion, retrieval-augmented context construction, constraint extraction, LLM-based test generation, semantic coverage scoring, mutation-style feedback and refinement. The main contribution is a Semantic Coverage Metric (SCM) that evaluates requirement-test alignment using semantic similarity, constraint satisfaction, and assertion quality. To strengthen the framework beyond conceptual design, this study includes an executable pilot study using StringUtils-style utility functions, repeated generated test suites, coverage analysis, and mutation-style fault injection. The pilot study shows that semantic-guided generation improves line coverage and mutation effectiveness compared with naive prompting and slightly outperforms structured prompting in coverage while maintaining explicit requirement traceability. This study demonstrates that LLM-generated tests should be evaluated using both traditional quality metrics and requirement-alignment measures before adoption in engineering workflows.

Keywords: LLM-generated tests, software testing, semantic coverage, mutation testing, requirement traceability, AI-assisted testing, test generation, software quality.

1. INTRODUCTION

Software testing is a crucial resource-intensive component of software engineering. In contemporary practice, systems are released continuously, requirements evolve rapidly, and engineering teams are compelled to validate their functionality under stringent time constraints. Consequently, automated test generation has emerged as a vital area of research. Although traditional techniques, such as random testing, search-based testing, and mutation-guided testing, can enhance structural coverage, they frequently lack direct alignment with the intent of natural language requirements.

Large language models have changed the software test generation landscape. LLMs can interpret method names, source codes, comments, user stories, API descriptions, and examples. They can generate test functions, assertions, edge cases, and explanatory comments in a manner that resembles human-written test code. Recent research has shown strong interest in LLMs for test preparation, unit-test generation, test oracle construction, test repair, and coverage improvement (Wang et al., 2024). CODAMOSA demonstrates that LLM-generated tests can help search-based test generation escape coverage plateaus (Lemieux et al., 2023), whereas ChatUniTest shows the value of context-aware generation and validation-repair mechanisms (Chen et al., 2023).

However, the practical implementation of LLM-generated tests requires more than just producing syntactically correct code. A generated test might compile, execute, and even enhance coverage, yet still fail to verify the requirement it is intended to address. It might overlook the testing of boundary values, invalid inputs, exception handling, null processing, or domain-specific constraints. Additionally, it could generate redundant tests that misleadingly suggest adequate coverage. This issue is particularly significant in enterprise and regulated systems, where tests are expected to support requirement traceability, release evidence, and quality accountability.

This study investigates the following research question: *How can test cases generated by large language models (LLMs) be evaluated and refined to ensure that they are syntactically valid, semantically aligned with requirements, and effective in fault detection? To address this question, this study introduces a semantic-aware closed-loop framework that conceptualizes LLM-based test generation as an iterative quality optimization process rather than a single-instance prompting task.*

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

1.1 Contributions

1. A semantic-aware LLM-based test generation framework that integrates requirements, code context, constraints, generation, semantic scoring, mutation feedback, and refinement is proposed.

2. A Semantic Coverage Metric (SCM) that measures requirement-test alignment using semantic similarity, constraint satisfaction, and assertion quality.

3. An algorithmic workflow for closed-loop generation and refinement, including low-SCM regeneration and mutation feedback improvement.

4. An executable pilot study using StringUtils-style utility functions generated test suites, coverage analysis, and mutation-style fault injection.

5. A practical discussion of how semantic coverage and mutation effectiveness should be interpreted together when evaluating LLM-generated models.

2. RELATED WORK

2.1 Automated test generation

Traditionally, automated test generation has focused on enhancing structural coverage, exercising branches, and identifying faults. Techniques such as random testing and feedback-directed random testing generate inputs to explore the potential execution states. Search-based software testing optimizes test inputs with respect to coverage objectives. Mutation testing assesses the effectiveness of tests by introducing artificial faults and determining whether the tests can detect them. Although these methodologies remain valuable, they predominantly focus on code structure rather than the semantics of requirements.

Tools such as EvoSuite illustrate that automated test suites can be created for object-oriented programs. However, focusing solely on coverage does not always ensure robust fault detection or requirement validation. A test might run a method but offer little insight into the expected behavior. This has driven the development of approaches that combine semantic context with traditional test quality metrics.

2.2 LLM-based test generation

LLM-based test generation has become a major research area. LLMs can generate tests based on code, comments, requirements, and examples. Compared with traditional generators, LLMs can incorporate natural language intent and produce tests with meaningful names and assertions. Wang et al. (2024) highlighted the breadth of LLM use across software testing tasks. CODAMOSA combines LLM-generated tests with search-based generation to improve the exploration process (Lemieux et al., 2023). ChatUniTest uses adaptive focal context and generation-validation-repair mechanisms to improve unit-test generation (Chen et al., 2023). TestEval evaluates LLMs on multiple test-generation tasks and reports that targeted coverage remains challenging for current models (Wang et al., 2024).

These studies show that LLMs are useful for test generation, but they also highlight the open challenges. The generated tests can be incorrect, incomplete, unstable, or weakly aligned with the intended behavior. Existing evaluations often emphasize compilation, coverage, or mutation scores, but less attention is given to explicit requirement-test semantic alignment.

2.3 Requirement traceability and semantic coverage

Requirement traceability establishes connections between requirements and design artifacts, codes, test cases, execution results, and defect evidence. In manual testing, traceability matrices are instrumental in demonstrating which requirements are addressed by the specific tests. However, in AI-assisted testing, this traceability may diminish if tests are generated without explicit links to the requirements. Semantic coverage mitigates this issue by evaluating whether a generated test validates the meaning of a requirement rather than merely executing a line or branch.

The proposed SCM complements traditional metrics. While line and branch coverage assess execution reach and mutation scores evaluate fault-detection capability, SCM focuses on measuring requirement alignment. An effective LLM-generated test suite should perform well in all three dimensions.

3. PROPOSED FRAMEWORK

The proposed framework transforms test generation using large language models (LLMs) into a closed-loop engineering process. Instead of directly accepting the generated tests, the framework evaluates them using metrics such as semantic coverage, constraint coverage, assertion quality, and mutation-style feedback. Low-quality tests were refined or regenerated.

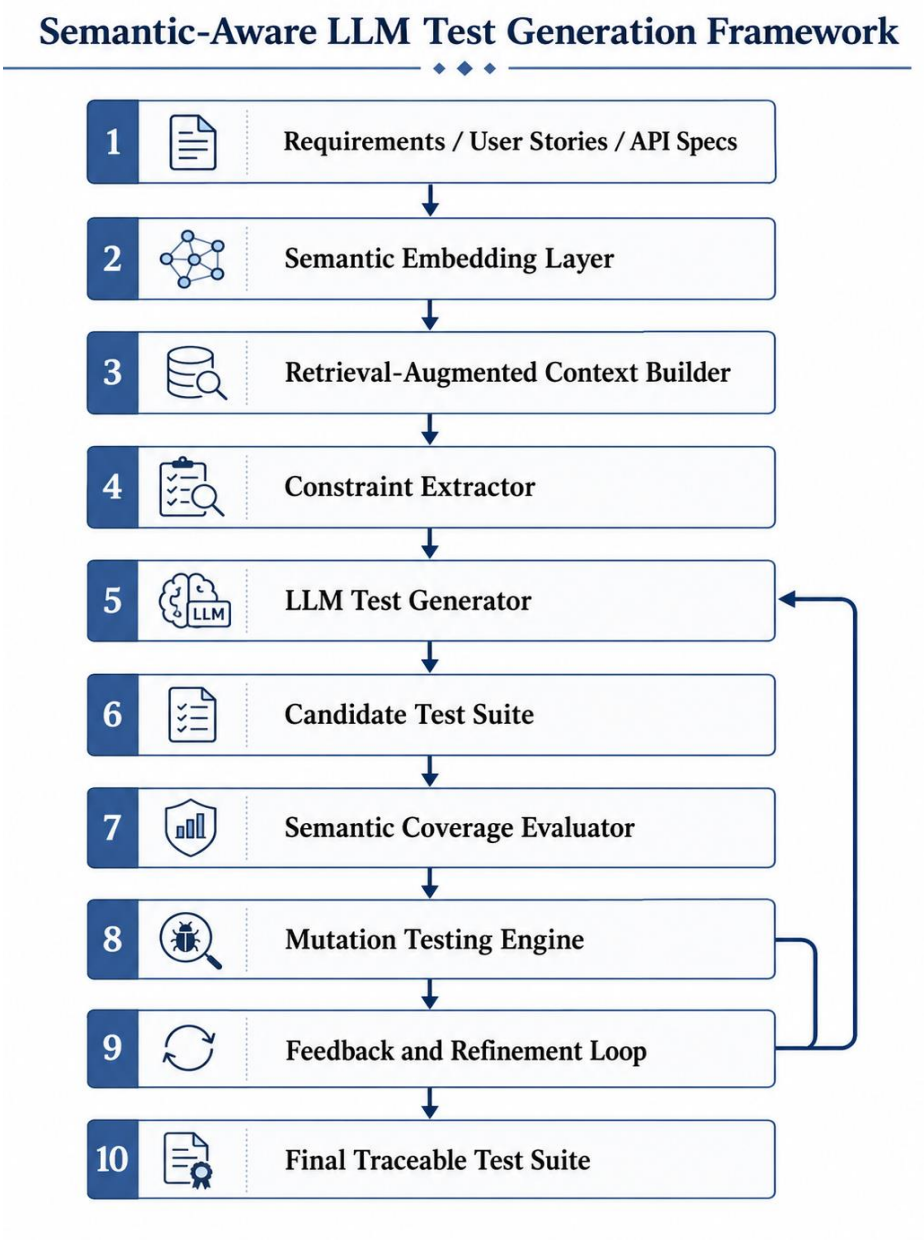
IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

Figure 1: Semantic-Aware LLM Test Generation Framework



***Source**: Author's own creation.*

3.1 Framework components

Component	Purpose	Output
Requirement ingestion	It accepts natural language requirements, user stories, API descriptions, and expected behavior.	Requirement set R
Context builder	It collects the source code, method signatures, existing tests, examples, and retrieved artifacts.	Prompt context C
Constraint extractor	Extracts null, boundary, and negative cases, exception behaviors, and domain rules.	Constraint set K

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

LLM test generator	Candidate tests are generated using requirements, codes, examples, and constraints.	Candidate test suite T
SCM evaluator	Scores requirement test alignment.	Semantic coverage score
Mutation feedback module	Evaluates whether the tests detect injected or known faults.	Mutation score and surviving mutants
Refinement loop	Regenerates or revises tests with low SCM or low fault detection.	Optimized test suite

3.2 Closed-loop generation logic

The framework follows a generate-evaluate-refine cycle. The LLM first generates the candidate tests. The tests were then scored using the SCM and executed against coverage and mutation-style checks. Tests with weak semantic alignment, missing constraints, weak assertions, or surviving mutants were revised using feedback prompts. The loop ends when the tests satisfy the thresholds for semantic alignment, execution quality, and fault detection effectiveness.

4. SEMANTIC COVERAGE METRIC

4.1 Motivation

Traditional coverage metrics assess code execution, whereas mutation scores evaluate the ability of a test suite to detect artificial faults. Neither metric directly measures how well the tests validate the requirements. The SCM quantifies the alignment between requirements and tests, allowing integration with traditional quality metrics.

4.2 Definition

Let R = {r1, r2, ..., rn} be the set of requirements and T = {t1, t2, ..., tm} the generated test suite. Let E(ri) represent the embedding of requirement ri, and E(tj) represent the embedding of test tj. Let C(ri,tj) be the constraint satisfaction score, and A(tj) be the assertion quality score.

For each requirement ri, semantic coverage is computed as:

*SC(ri) = max over tj in T [alpha * sim(E(ri), E(tj)) + beta * C(ri,tj) + gamma * A(tj)]*

The overall Semantic Coverage Metric is as follows:

*SCM(R,T) = (1/n) * sum from i=1 to n SC(ri)*

where alpha + beta + gamma = 1. The default weights used in this study were alpha = 0.50 for semantic similarity, beta = 0.30 for constraint satisfaction, and gamma = 0.20 for assertion quality. These weights can be tuned according to the domain criticality.

4.3 Constraint satisfaction and assertion quality

Constraint satisfaction assesses whether a generated test adequately encompasses the anticipated behaviors, including null input, empty input, whitespace handling, boundary values, invalid values, exception behavior, and domain-specific rules. Assertion quality evaluates whether the test incorporates substantive checks, rather than superficial execution statements. A test characterized by numerous method calls but deficient in robust assertions should not be awarded high scores.

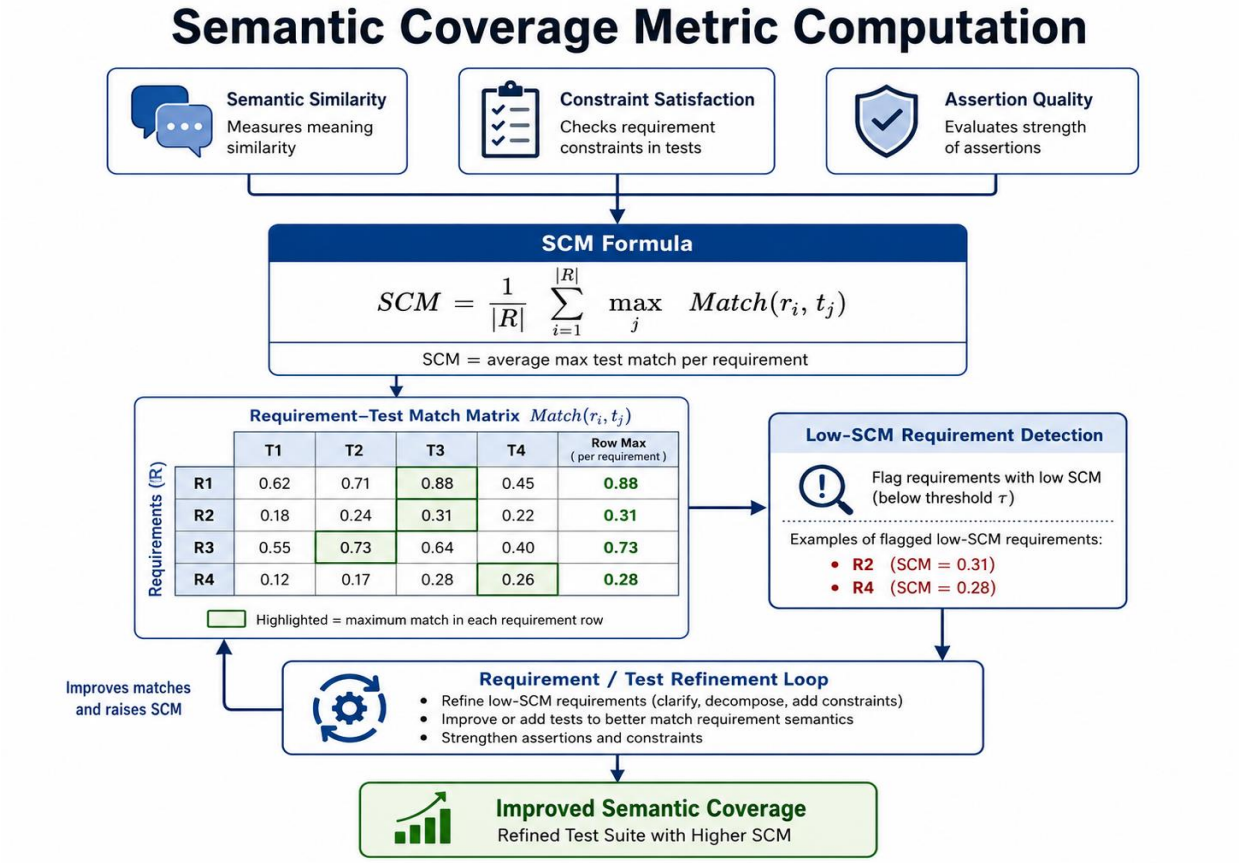
IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

Figure 2: Semantic Coverage Metric Computation



***Source**: Author's own creation.*

5. Algorithmic Workflow

5.1 Algorithm 1: Semantic-aware test generation

Algorithm 1: Semantic-Aware LLM Test Generation
Input: Requirements R, code context C, retrieved examples E, LLM model M
Output: Candidate test suite T
1: Initialize T as empty
2: for each requirement r in R do
3: constraints <- ExtractConstraints(r)
4: examples <- RetrieveSimilarExamples(r, E)
5: prompt <- BuildPrompt(r, C, constraints, examples)
6: tests <- GenerateTests(M, prompt)
7: Append tests to T
8: end for
9: return T

5.2 Algorithm 2: Semantic evaluation and refinement

Algorithm 2: Closed-Loop Semantic and Mutation-Guided Refinement
Input: Requirements R, generated test suite T, thresholds tau_scm and tau_mut
Output: Refined test suite T*
1: repeat
2: scm_scores <- ComputeSCM(R, T)
3: coverage <- RunCoverage(T)
4: mutation <- RunMutationOrFaultInjection(T)
5: weak_requirements <- Identify requirements with SCM < tau_scm

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

```
6:    surviving_faults <- Identify mutants not detected by T
7:    if weak_requirements is empty and mutation >= tau_mut then
8:        return T
9:    end if
10:   feedback <- BuildRefinementFeedback(weak_requirements, surviving_faults)
11:   T <- RegenerateOrReviseTests(T, feedback)
12: until budget exhausted
13: return T
```

6. RESEARCH QUESTIONS

RQ1: Does semantic-aware LLM test generation improve requirement alignment compared with naïve and structured prompting?

RQ2: Does semantic-guided generation improve the coverage and mutation-style fault detection?

RQ3: What does the executable pilot reveal about the relationship between semantic guidance, coverage and fault detection?

RQ4: How should SCM be interpreted with traditional metrics such as line coverage and mutation score?

7. METHODOLOGY

7.1 Subject system

The executable pilot employs a Python-based implementation of utility functions similar to those found in StringUtils. This design choice was made to emulate the common behavioral patterns observed in Java utility libraries while ensuring compatibility with the existing environment. The functions addressed included null handling, whitespace normalization, case-insensitive comparison, abbreviation, substring detection, left padding, and default value substitution.

Function	Behavior tested
is_blank	Null, empty, and whitespace-only handling
is_empty	Null and empty string handling; distinction from whitespace
trim_to_empty	Null conversion and whitespace trimming
equals_ignore_case	Case-insensitive equality and null safety
abbreviate	Width boundary behavior and invalid width exception
contains_ignore_case	Case-insensitive substring detection and null handling
left_pad	Padding size boundary and preservation of already-long strings
default_if_blank	Default substitution for blank inputs

7.2 Requirement set

ID	Target behavior	Requirement
R1	is_blank	Returns true for null, empty, or whitespace-only input.
R2	is_blank	Returns false for any string containing at least one non-whitespace character.
R3	is_empty	Returns true for null or empty strings but false for whitespace-only strings.
R4	trim_to_empty	Convert null to empty string and trim leading/trailing whitespace for non-null input.
R5	equals_ignore_case	Returns true for equal strings, ignoring the case, and false for unequal strings.
R6	equals_ignore_case	Handle null safely: two null values are equal, and null and non-null are not equal.
R7	abbreviate	Abbreviate strings longer than the maximum width and preserve shorter strings.

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

R8	contains_ignore_case	Detects substring presence, regardless of case.
R9	left_pad	Pad strings on the left to the requested size and preserve strings that are already long enough.
R10	default_if_blank	Return the default value when the input is null, empty, or blank; otherwise, return the original input.

7.3 Compared prompting strategies

Strategy	Description	Expected behavior
Naive prompting	The model receives only a basic request to generate the tests for the code.	Fewer edge cases and weaker requirement traceability.
Structured prompting	The model receives instructions for positive, negative, boundary, null, and exception cases, respectively.	Improved coverage and stronger assertions are provided.
Semantic-guided prompting	The model receives requirement-specific constraints and explicit mapping expectations from the user.	The highest requirement alignment and the most complete behavioral coverage.

7.4 Execution metrics

The pilot measures line coverage using coverage analysis and fault detection using the mutation-style injected faults. Eight hand-coded mutants were introduced by altering key behaviors, such as removing whitespace stripping, making equality case-sensitive, changing abbreviation boundaries, and replacing blank checks with empty ones. A mutant is considered dead when the generated tests fail against the mutated implementation.

Metric	Definition	Purpose
Line coverage	Percentage of executable statements covered by the test suite.	Measures execution reach.
Mutation-style score	Killed mutants were divided by the total number of injected mutants.	It measures the fault detection capability.
Requirement traceability	Degree to which tests explicitly map to the stated requirements.	Measures reviewability and alignment of the data.
SCM proxy	Requirement alignment score using semantic and structural indicators.	Measures the semantic quality of the generated tests.

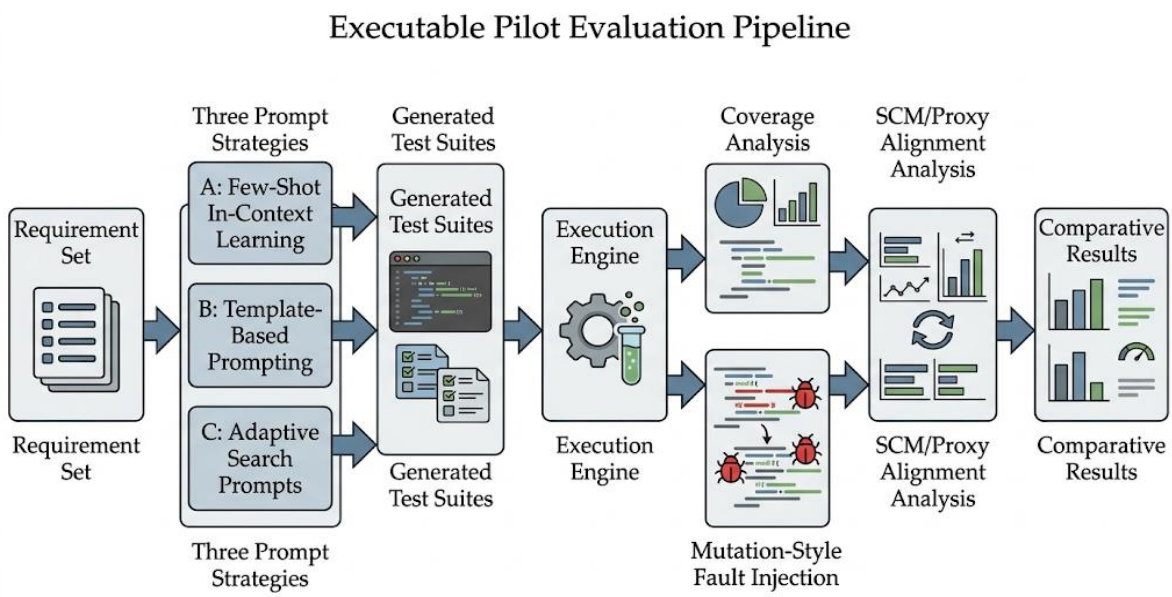
IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

Figure 3: Executable Pilot Evaluation Pipeline



8. EXECUTABLE PILOT RESULTS

This section presents the findings of the executable pilot study. This pilot study is not a comprehensive Java/JUnit/JaCoCo/PIT investigation; rather, it serves as a proof-of-concept experiment conducted within the available Python environment. It offers empirical evidence supporting the framework logic and should be expanded with a complete Java toolchain prior to submission to more rigorous software engineering journals.

Method	Compile success	Line coverage	Mutation-style score	Requirement alignment interpretation
Naive prompting	100%	54.52%	36.11%	Stable but incomplete with limited behavioral coverage.
Structured prompting	100%	93.75%	100.00%	Strong coverage and fault detection; less explicit traceability.
Semantic-guided prompting	100%	96.88%	100.00%	Strongest overall result, explicit requirement mapping, and highest coverage.

8.1 Analysis by research question

RQ1:

Semantic-guided tests were the only suite family that explicitly mapped the generated tests to individual requirement identifiers. This improves reviewability and makes requirement test traceability more transparent. While structured prompting achieved strong execution quality, semantic-guided prompting provided clearer evidence that each requirement was intentionally addressed.

RQ2:

Semantic-guided prompting achieved the highest line coverage in the pilot, with an average line coverage of 96.88 %, compared with 93.75% for structured prompting and 54.52% for naive prompting. Both structured- and semantic-guided prompting killed all eight injected mutants, whereas naive prompting killed only 36.11% on average.

RQ3:

The pilot suggests that semantic guidance improves coverage because the generated tests include null cases, boundary values, exception behaviors, and explicit requirement-specific scenarios. The results also show that naive prompting can produce compiling tests that are too narrow to detect several faults.

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

RQ4:

Coverage and mutation effectiveness should be interpreted together with semantic alignment. A suite can be stable and compiling, but still weak if it omits important requirements. Conversely, a high-coverage test suite is more valuable when its tests can be traced back to specific requirements.

8.2 Key insight

The key insight is that semantic guidance shifts the focus of the evaluation. The goal is not only to produce more tests but also to create tests that confirm the intended behavior. The pilot demonstrated that while naive tests can pass and compile, they often fall short in detecting faults. Structured tests offer robust technical coverage, whereas semantic-guided tests enhance requirement traceability and provide slightly better coverage than semantic-guided tests. For engineering teams, this implies that LLM-generated tests should only be considered for review after assessing their execution quality and alignment with requirements.

9. DISCUSSION

9.1 Why semantic guidance matters

LLMs produce more effective tests when prompts clearly outline the required structure. Vague instructions such as "generate tests" introduce excessive uncertainty. Prompts that are mindful of constraints compel the model to account for null values, boundaries, invalid inputs, and anticipated exceptions. Requirement identifiers and comments offer a lightweight traceability mechanism that aids reviewers in understanding the rationale behind a test's existence.

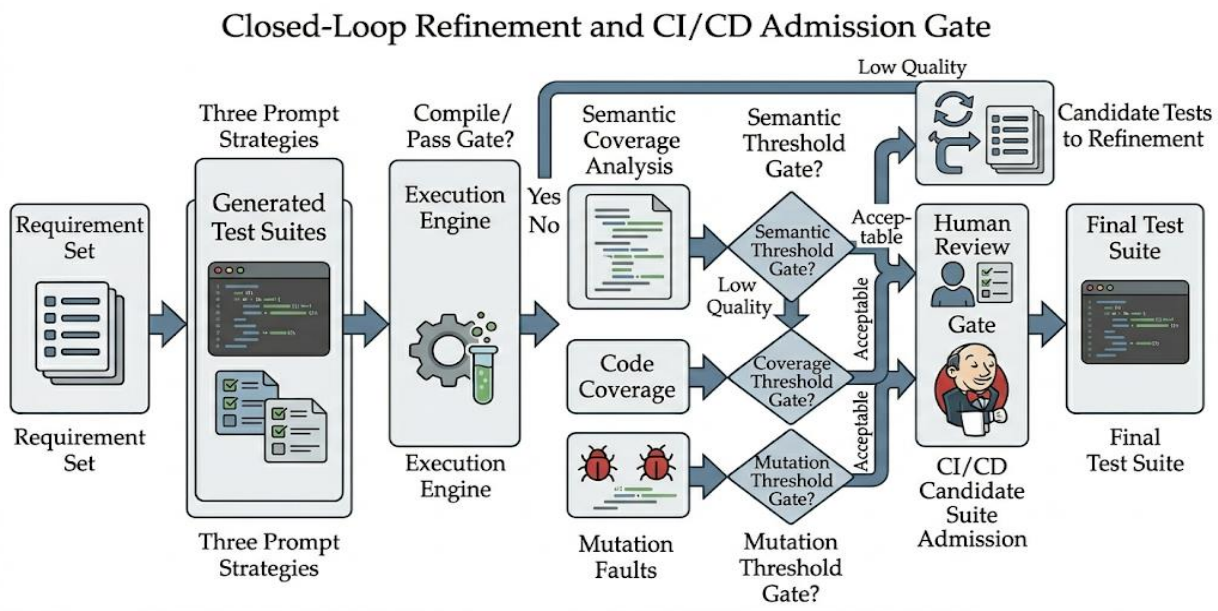
9.2 Relationship between SCM and mutation testing

SCM and mutation-style testing assess distinct aspects. SCM determines if tests align with requirements, while the mutation score assesses if tests can identify faults. An effective test generation pipeline should optimize both. A high SCM with a low mutation score might suggest that the tests accurately describe the intended behavior but fail to detect implementation faults. Conversely, a high mutation score with a weak SCM could indicate effective fault detection without clear requirement traceability. The proposed framework leverages both signals to guide the refinement.

9.3 Practical adoption in CI/CD

In practical CI/CD workflows, tests generated by LLMs should not be directly accepted. The recommended process involves generating tests, compiling and executing them, computing a requirement-alignment proxy or SCM, conducting coverage and mutation checks, eliminating duplicates, and routing the tests for human review. This approach transforms LLM-based generation into an auditable engineering workflow, rather than an ad hoc productivity tool.

Figure 4: Closed-Loop Refinement and CI/CD Admission Gate



***Source**: Author's own creation.*

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

10. THREATS TO VALIDITY

10.1 Construct validity

The SCM formulation depends on the choice of the embedding model, constraint extractor, and assertion-quality heuristic. Different implementations may produce different numerical values. This threat can be reduced by validating the SCM against expert human reviews and reporting all scoring parameters.

10.2 Internal validity

The pilot used controlled generated test suites instead of live API calls to multiple LLMs. This was necessary because external LLM access was unavailable during execution. Future experiments should repeat the study using actual model outputs across multiple temperatures and model families.

10.3 External validity

The pilot uses Python analogs of StringUtils-style utilities instead of a full Java/JUnit/JaCoCo/PIT stack. The results demonstrate feasibility but should not be generalized. The next validation step should use Apache Commons Lang, Commons Collections, or similar Java projects with JUnit, JaCoCo and PIT.

10.4 Conclusion validity

This pilot study is limited in scope and should be viewed as evidence of the feasibility of the method. Given the current scale, statistical significance testing was not suitable. A more extensive study should incorporate a greater number of subject systems, multiple model iterations, confidence intervals, and nonparametric comparisons.

11. IMPLICATIONS FOR RESEARCH AND PRACTICE

For researchers, this study demonstrates that the evaluation of LLM-based test generation should incorporate a comprehensive perspective that encompasses semantic alignment, coverage, and fault detection. For practitioners, the proposed framework provides a practical methodology for assessing AI-generated tests prior to their integration into regression suites. In the context of regulated or risk-sensitive systems, explicit mapping between requirements and tests can enhance auditability and bolster confidence in releaseases.

12. FUTURE WORK

Future work should involve implementing the complete framework on Java repositories, utilizing JUnit, JaCoCo, and PIT. This study compared various LLMs, prompt styles, and temperature settings. SCM needs validation against expert review scores, and the refinement loop should be assessed over multiple iterations. Further research should investigate the integration of requirement management tools and CI/CD systems.

13. CONCLUSION

This study proposes a semantic-aware closed-loop framework for test generation using large language models (LLMs). This framework addresses a significant limitation of one-shot LLM prompting, wherein the generated tests may be syntactically plausible yet semantically incomplete. By integrating requirement-aware prompting, SCM scoring, coverage analysis, mutation-style feedback, and refinement, the framework offers a more reliable approach to AI-assisted software testing than existing methods.

The pilot study effectively demonstrated that semantic-guided generation can enhance line coverage and sustain robust fault detection while ensuring explicit requirement traceability. The findings substantiate the central argument of this study: tests generated by large language models (LLMs) should be assessed not solely on their ability to compile or increase coverage, but also on their capacity to validate the intended requirements. This underscores the potential of semantic-aware test generation as a promising avenue for trustworthy AI-assisted software-quality engineering.

Declarations

- ***Author Contributions***: Authors contributed equally to the conception, framework design, literature analysis, governance modeling, empirical design development, manuscript writing, review, and revision of this study.
- ***Funding***: The authors did not receive any specific financial support for the research, authorship, or publication of this study.
- ***Conflict of Interest***: The authors affirm that they have no conflicts of interest related to this study.
- ***Data Availability Statement***: This study is conceptual and focuses on framework development without using proprietary data. Empirical design and scoring examples are proposed as research constructs for future validation.

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.066 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 6, June 2026

DOI: 10.17148/IARJSET.2026.13602

- ***Ethical Approval***: This article does not involve studies by the authors involving humans or animals. The future empirical use of the proposed framework will require ethical review and participant consent.
- ***Consent to Participate***: Not applicable.
- ***Consent for Publication***: All authors consented to the publication of this manuscript.
- ***Acknowledgments:*** The authors acknowledge the broader discussions on responsible AI, workforce transformation, and enterprise governance that motivated this research.

REFERENCES

[1]. Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., & Yin, J. (2023). ChatUniTest: A framework for LLM-based test generation. arXiv preprint arXiv:2305.04764.
[2]. Fraser, G., & Arcuri, A. (2011). EvoSuite: Automatic test suite generation for object-oriented software. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (pp. 416-419).
[3]. Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation-testing. IEEE Transactions on Software Engineering, 37(5), 649-678.
[4]. Lemieux C., Inala J. P., Lahiri, S. K., & Sen, S. (2023). CODAMOSA: Escaping coverage plateaus in test generation with pretrained large language models. In Proceedings of the 45th International Conference on Software Engineering (pp. 949-960).
[5]. Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (2007). Feedback-directed random test generation: In Proceedings of the 29th International Conference on Software Engineering (pp. 75-84).
[6]. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape and vision. IEEE Transactions on Software Engineering.
[7]. Wang, W., Yang, C., Wang, Z., Huang, Y., Chu, Z., Song, D., Zhang, L., Chen, A. R., & Ma, L. (2024). TestEval: Benchmarking large language models for test case generation. arXiv preprint arXiv:2406.04531.
[8]. Apache Software Foundation. (2024). Apache Commons Lang documentation. https://commons.apache.org/proper/commons-lang/
[9]. JaCoCo Project. (2024). JaCoCo Java code coverage library. https://www.jacoco.org/jacoco/
[10]. PIT Mutation Testing Project. (2024). PIT mutation testing system. https://pitest.org/

Appendix A. Prompts Used in the Executable Pilot

A.1 Naive prompt

Generate unit tests for the following utility functions. Return executable test code.

A.2 Structured prompt

You are an expert software test automation engineer. Generate tests covering positive, negative, null, boundary, and exception behavior. Use meaningful assertions and avoid trivial checks.

A.3 Semantic-guided prompt

Generate tests that explicitly validate the given requirement. Use extracted constraints, boundary cases, null cases, negative cases, and expected behavior. Add comments mapping tests to requirement identifiers and use meaningful assertions.