International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 10 | Oct 2026 www.irjet.net p-ISSN: 2395-0072

Validation and Repair of LLM-Generated Unit-Test Oracles: A Focused Review of Reliability Interventions and Fault-Grounded Evaluation

Purav Salla[1], Pratiksha Navle[2], Priyesh Mandhalkar[3], Dr Ashwini Kadam[4]

[1234]*Department of MCA, D. Y. Patil University, School of Humanities and Sciences, Navi Mumbai, India*

Abstract - *Large language models (LLMs) can generate unit-test assertions and other test oracles, but syntactically plausible output does not guarantee that the oracle represents intended software behaviour or detects faults. This paper presents a focused systematic literature review of methods used to validate, repair, ground, and evaluate LLM-generated unit-test oracles. The review synthesizes eleven primary studies published or formally accepted during 2025–2026, while using relevant 2024 work to contextualize evaluation practice. Studies were coded by reliability failure, intervention, source of behavioural authority, benchmark characteristics, and evaluation strength. The synthesis identifies six recurring concerns: non-compiling output, execution failures and false positives, weak fault-detection oracles, missing or misleading context, misguidance from buggy implementations, and unrealistic benchmark assumptions. Five intervention families emerge: static/dynamic feedback, retrieval-and-repair, prompt and context engineering, documentation/specification grounding, and realistic benchmark design. Reported evidence illustrates why exact match is insufficient: a 2024 study found no significant correlation between textual-similarity metrics and test-adequacy metrics; a 2025 unbiased-dataset study reported average mutation scores of 43% for LLM generated oracles and 45% for human-designed oracles; and TOGBench (2026) evaluates generated oracles in runnable developer-written tests. Because the reviewed studies use heterogeneous models, datasets, and metrics, this review does not calculate a pooled effect size or claim new statistical significance. It recommends layered, contamination-aware evaluation that measures compilation, execution, false positives, real-defect exposure, and mutation-based strength.*

Key Words: Large language models; software testing; test oracle generation; assertion generation; mutation testing; reliability; systematic literature review.

1.INTRODUCTION

A software test is useful only when it can distinguish acceptable program behaviour from incorrect behaviour. The component that makes this decision is the test oracle. In unit testing, an oracle may be an assertion over a return value or object state, a check for an exception, or a more complex property. The oracle problem arises because the intended result is often difficult or expensive to specify explicitly [1]. Traditional automated test-generation systems can produce inputs at scale, but their oracles may be based on existing program behaviour or simple rules and may therefore preserve defects rather than expose them. LLMs offer a flexible way to generate assertions from source code, test prefixes, method signatures, examples, comments, documentation, and specifications. Yet the generated assertion can compile and execute while still checking the wrong value. If the model relies only on a buggy implementation, it may reproduce the bug as expected behaviour. Similarly, a generated assertion may be too weak to detect meaningful faults or too strict and fail on correct software. These failure modes make reliability a multi-dimensional property rather than a single accuracy score. This paper focuses on the reliability of LLM-generated unit-test oracles rather than general test generation. It reviews recent validation and repair methods, examines the authority used to infer expected behaviour, compares evaluation strategies, and identifies a practical evidence hierarchy for research and industry. running text should match with the list of references at the end of the paper.

1.1 Problem Statement

The central problem is that current evaluations can overestimate the reliability of LLM-generated unit-test oracles when they emphasize exact textual match or compilation without verifying whether the oracle accepts correct software and rejects faulty software. A generated oracle may differ textually from a reference oracle yet be semantically equivalent; conversely, an exact-looking assertion may be weak, misleading, or unable to expose a fault. A further risk occurs when the implementation under test contains a defect and the model uses that implementation as its main source of expected behaviour. The research problem is therefore to understand which reliability failures are reported, which interventions mitigate them, and what evidence is needed to judge the resulting oracles as trustworthy. The review also considers whether benchmark design reflects developer written tests, mixed assertion patterns, realistic insertion positions, and contamination-aware evaluation.

1.2 Objectives

1. Develop a reliability taxonomy grounded in observable failures reported by recent studies.

2. Classify validation, repair, and grounding interventions and map them to the failures they address.
3. Compare evaluation metrics, datasets, and benchmark assumptions, with special attention to fault-grounded evidence.
4. Synthesize recent evidence from 2024–2026 without combining incompatible metrics into a misleading pooled estimate.
5. Formulate testable hypotheses and specify a transparent statistical-testing plan for future primary experimentation.
6. Provide recommendations for researchers and practitioners adopting

1.3 Research Questions

RQ1 What reliability failures are reported in LLM-generated unit-test oracles?
RQ2 Which techniques are used to validate, repair, or ground generated oracles?
RQ3 How do prompt context, retrieval, documentation, and specifications affect reported quality?
RQ4 Which datasets and metrics are used, and how realistic are their evaluation settings?
RQ5 To what extent do studies measure real-defect exposure, false positives, or mutation-based fault detection?
RQ6 What methodological gaps and practical risks remain after current interventions?

2. Hypotheses

H1 (evaluation validity): Textual similarity/exact-match scores alone are insufficient proxies for fault-detection strength. H0I: textual similarity metrics are adequate proxies for test adequacy across evaluated oracle-generation systems; H1I: they are not adequate proxies. H2 (external evidence): Adding independent behavioural evidence, such as reliable documentation or specifications, is associated with improved oracle correctness or fault detection compared with relying on implementation context alone. H0I: independent behavioural evidence does not improve these outcomes; H1I: it improves at least one outcome. H3 (feedback repair): Combining LLM generation with static analysis and execution feedback improves mechanical validity compared with one-shot generation. H0I: feedback does not improve compilation/execution validity; H1I: feedback improves at least one validity measure. These are review-derived hypotheses, not claims that this paper ran a new controlled experiment. The evidence is assessed through reported findings and methodological comparison. Inferential testing requires compatible paired observations or raw benchmark outputs; where such data are unavailable, no p-values are fabricated.

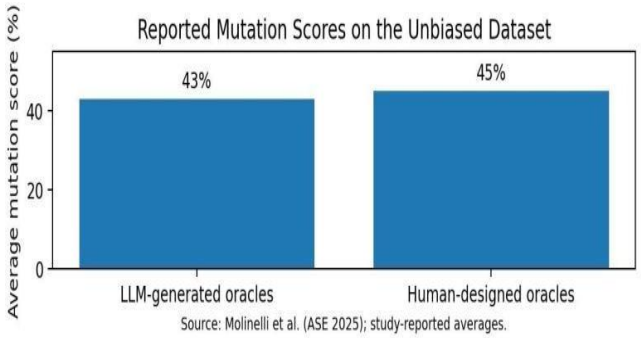
3. Literature Review: 2024–2025

Research published in 2024 established an important measurement concern. Shin et al. [2] evaluated neural test-oracle generation using seven textual-similarity metrics and two test-adequacy metrics. Their analysis reported no significant correlation between the textual similarity and test-adequacy metrics in the studied setting. This matters because exact match, BLEU-like scores, or other text comparisons can reward resemblance without demonstrating that an oracle kills faults. It directly motivates H1 and a layered evaluation design. Hayet et al. introduced ChatAssert, a framework that combines LLM prompting with static and dynamic tools [3]. Candidate assertions are generated with supporting context, checked by lightweight static analysis, and refined using test-execution feedback. The study reports a 15-percentage-point improvement in Acc@1 over TECO on its evaluated dataset. This work represents a shift from treating the LLM as the final authority to treating it as one component in a generate–validate–repair loop. The improvement is tied to the evaluated dataset and should not be generalized automatically to all languages or projects. Zhang et al. [4] investigated assertion generation with multiple LLMs and retrieval-and-repair. Their results report an average additional prediction-accuracy improvement of 9.40% from the retrieval-and-repair formulation, and the study also considers defect detection on Defects4J. Retrieval can provide a useful structural example, but an example must be adapted to the target method and cannot independently prove that its expected value is correct. Hossain and Dwyer's TOGLL work [5] targets correct and strong oracles through fine-tuning and prompting. In addition to correctness, the work considers diversity and fault-detection outcomes, illustrating that a test oracle should be judged by whether it can expose defects. Documentation-oriented approaches also became prominent. AugmenTest [6] uses developer comments and documentation as evidence for oracle generation. Doc2OracLL [7] studies Javadoc as a behavioural source and reports that including 2 Javadoc comments improves ground-truth oracle matching by approximately 10–20% in its evaluated settings; the description and @return information are particularly useful. These findings support grounding expected behaviour in artifacts that can be independent of implementation details. Molinelli et al. [8] address contamination risk by using 13,866 oracles from 135 Java projects created after the evaluated LLMs' training cutoffs. They report average mutation scores of 43% for LLM-generated oracles and 45% for human-designed oracles. They also report that test prefixes and called methods can provide useful context, while additional code context does not necessarily improve results. This suggests that more prompt material is not always better: relevance, authority, and contamination control matter. Bodicoat et al. [9] examine prompt and context sensitivity. Their reported results show that including the class-under-test context improved compilation rate from 49.36% to 76.26% and

overall accuracy from 40.74% to 53.64% in their evaluated setting. These results should be interpreted as context specific evidence rather than a universal rule that more context always helps. Across these studies, the recurring methodological lesson is to report both generation quality and executable, fault-grounded outcomes.

4. Literature Review: 2026 and Comparative Synthesis

TOGBench, introduced by Tasnim et al. [10], responds to limitations in simplified oracle-generation benchmarks. Its OE25dev dataset is curated from developer-written tests across 25 open-source Java projects and 56 modules. It preserves six oracle categories and realistic configurations including single and multiple oracles, mixed assertion-and-exception checks, and custom oracles. The benchmark reintegrates generated oracles into runnable tests and evaluates compilation, execution, false positives, and mutation testing. The authors report that representative methods experience marked performance degradation on developer-written tests, particularly for assertion oracles. Fu et al. [11] propose specification-driven test-oracle enhancement, combining specification reasoning with repair and adjudication. The supplied review reports evaluation on 3,961 Java SE API test cases and 45 previously unknown OpenJDK bugs confirmed or fixed by developers, alongside 88.9% precision for the reporting pipeline. These are study-reported figures, not results reproduced by the present review. Their practical value lies in moving beyond benchmark similarity toward defects that developers can confirm. Li et al. [12] study assertion generation under mixed-assertion scenarios. In 358,117 developer-written tests from 7,061 projects, 40.32% contained multiple assertions, and such tests appeared in 92.87% of projects. Removing known insertion-position cues reduced exact match by 11.80–23.18%. Their two-stage DA-AG framework improved exact match by 32.24–78.08% over corresponding one-stage baselines and increased Defects4J bug exposure. These results show that benchmark assumptions about one assertion per test and known insertion positions can materially affect measured performance. Zhao et al. [13] examine the misguidance effect of buggy code. Their specification-based mitigation seeks to reduce the risk that an LLM encodes the defect in the implementation as expected behaviour. This is a distinct semantic failure: the output may be syntactically valid and may pass the buggy implementation, but it is still an incorrect oracle for the intended contract. Study / year Approach or dataset Evaluation emphasis Key reported implication Shin et al., 2024 [2] Neural oracle metrics Text similarity vs. test adequacy Similarity was not significantly correlated with adequacy in the studied setting. ChatAssert, 2025 issue Static + dynamic repair Compilation, execution, Acc@1 Tool feedback can repair mechanical failures. [3] Doc2OracLL, 2025 [7] Javadoc grounding Exact match and bug detection Behavioural documentation can improve oracle generation. Molinelli et al., 2025 Post-cutoff dataset Mutation score 43% LLM vs. 45% human average mutation [8] score. TOGBench, 2026 [10] Developer-written multi-variant tests Compile, execute, false positives, Realistic test structure reveals weaknesses. mutation Li et al., 2026 [12] Mixed assertions / unknown positions Exact match, CodeBLEU, bug exposure Simplified insertion assumptions inflate apparent performance. Across the literature, datasets and methods are heterogeneous. The figures above are therefore compared descriptively, not treated as directly interchangeable measurements.



5. Research Gap

Gap 1 — No universally adopted, end-to-end benchmark: Recent work improves realism, but the field still lacks one widely adopted benchmark that jointly controls training-data contamination, preserves developer-written test structure, supports diverse oracle types, and provides executable correct/faulty program pairs.

Gap 2 — Residual risk after repair is inconsistently reported: A repaired oracle may compile yet remain semantically wrong or too weak. Studies need to report stage-by-stage survival rates from generation through compilation, execution, false-positive control, fault exposure, and mutation adequacy.

Gap 3 — Metrics remain difficult to compare: Exact match, prediction accuracy, compilation rate, mutation score, and real-bug detection measure different properties. A pooled statistical result is inappropriate unless datasets, units of analysis, and definitions are harmonized.

Gap 4 — Language and domain coverage: Much of the strongest evidence is Java-centric. Generalization to Python, JavaScript, C/C++, dynamic typing, integration tests, and industrial codebases remains uncertain.

Gap 5 — Specification quality is an upstream dependency: Documentation and specifications can reduce implementation bias, but inaccurate, stale, incomplete, or generated specifications may themselves mislead the model. Future evaluation should measure agreement among specification-derived, implementation-derived, and human-reviewed expected behaviour.

6. Research Methodology

6.1 Design

This study uses a focused systematic literature review (SLR), following the software-engineering review principles of Kitchenham and Charters [14] and using PRISMA 2020 as a reporting reference [15]. It is explicitly a focused review based on accessible scholarly records and verified publication information, not a claim of exhaustive database-level screening.

6.2 Scope and eligibility

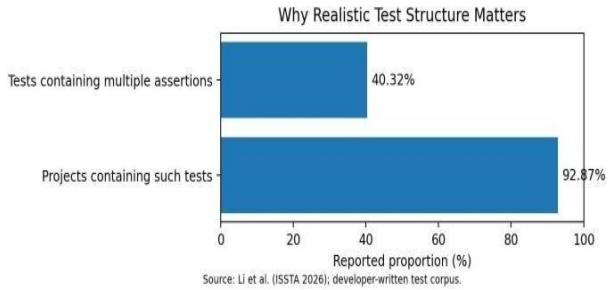
The primary evidence set covers eleven studies published or formally accepted in 2025–2026 that directly address LLM-generated unit-test assertions/oracles and evaluate validation, repair, grounding, or oracle reliability. A 2024 evaluation-metrics study is included as contextual evidence. Secondary reviews and classic oracle surveys are used for background but are not counted among the eleven primary studies.

6.3 Search strategy

Search concepts combine (i) LLM terms (“large language model”, LLM, GPT, generative AI), (ii) testing terms (unit testing, automated testing, test generation), and (iii) oracle terms (test oracle, assertion generation, test assertion). Publication records were checked using publisher and conference pages, IEEE Xplore, ACM Digital Library, DBLP, and accessible author/institutional copies. Reference-list checking was used to identify related studies.

6.4 Selection and extraction

After title-level deduplication and relevance screening, fourteen highly relevant records were retained; three were treated as background sources, leaving eleven primary studies. For each primary study, the review records year, task, intervention, source of context/authority, language and benchmark where reported, evaluation metrics, reported results, and stated limitations. The evidence is synthesized qualitatively because the underlying samples, models, metrics, and experimental designs are not sufficiently homogeneous for a defensible pooled meta-analysis.



7. Data Extraction, Analysis and Tools

7.1 Coding framework

Each study is coded against five evidence levels: (1) textual/reference similarity; (2) compilation and type validity; (3) execution validity and false-positive behaviour;
(4) real-defect exposure on buggy/fixed versions; and (5) mutation-based fault detection or developer-confirmed defects. These levels are ordered by the strength of the claim they can support, not by the convenience of calculating the metric. Reliability level Metric / evidence What it establishes What it cannot establish alone L1: textual Exact match, CodeBLEU, similarity Resemblance to a reference Semantic correctness or fault detection L2: compilation Compile rate, type checking Syntactic/type compatibility Correct expected behaviour L3: execution Pass/fail on correct code; false-positive Runtime behaviour and some semantic General fault-detection strength rate validity L4: real defects Bug exposure on buggy/fixed versions Ability to distinguish known faulty Broad generalization beyond the benchmark behaviour L5: Mutation score, confirmed defect Fault sensitivity under a defined All possible real-world failures mutation/confirmed reports protocol bugs

7.2 Analysis method

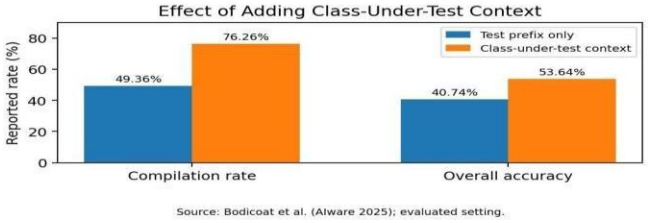
A descriptive cross-study matrix is used to map reliability failures to interventions. Numerical values are retained with the study that reported them and are not averaged across incompatible denominators. The synthesis compares direction and type of evidence, benchmark realism, and the degree to which each method independently grounds expected behaviour.

7.3 Tools and technologies

The review is based on scholarly search and publisher/conference metadata and uses a structured extraction sheet for coding. The reviewed technical systems use LLM prompting or fine-tuning, Java test suites, static analysis, test execution, documentation/specifications, retrieval, Defects4J, and mutation testing where reported. For a reproducible follow-up experiment, a practical stack would include Python for data preparation and statistical analysis, a version-controlled extraction spreadsheet, Java build/test tooling, Defects4J or TOGBench-compatible artifacts, and a statistical package such as SciPy/R. These are proposed tools for replication, not a claim that every tool was executed in this literature review.

8. Data Analysis and Interpretation

The evidence supports a layered view of reliability. Static and dynamic feedback target non-compiling or non-passing candidates. Retrieval-and-repair can improve adaptation from examples. Prompt/context engineering can improve results when relevant context is selected. Documentation/specification grounding helps separate intended behaviour from the current implementation. Realistic benchmark redesign tests whether methods survive the complexity of developer-written suites. The review identifies six recurring failure families: (i) syntax/type errors; (ii) execution failures and false positives; (iii) weak oracles that pass faulty code; (iv) missing, irrelevant, or hallucinated context; (v) buggy-code misguidance; and (vi) benchmark unrealism. The intervention mapping is not one-to-one: for example, specification grounding may reduce implementation bias but cannot guarantee that the specification is accurate; execution feedback can remove failing assertions but can also reinforce a buggy implementation if that implementation is treated as ground truth. The 43% versus 45% mutation-score comparison reported by Molinelli et al. [8] is evidence that generated oracles can approach human designed oracles on a defined unbiased dataset, but it is not evidence that every generated oracle is reliable. Similarly, reported gains in exact match under DA-AG [12] must be interpreted alongside realistic insertion-position assumptions and bug-exposure results. The strongest conclusions arise when independent measures converge.



Source: Bodicoat et al. (AIware 2025); evaluated setting.

9. Hypothesis Assessment and Statistical Testing Plan

H1 assessment. The reviewed evidence is consistent with H1. Shin et al. [2] report no significant correlation between textual-similarity and test-adequacy metrics in their evaluated setting, while later studies emphasize mutation testing and real-defect exposure. This supports the argument that textual resemblance alone is inadequate. It does not establish a universal statistical relationship across all LLMs, languages, and datasets.

H2 assessment. The reviewed studies provide descriptive support for H2. Doc2OracLL [7] reports gains from Javadoc information and improved bug detection in its evaluated settings; specification-driven approaches [11], [13] aim to reduce implementation bias. However, the studies use different models, tasks, and benchmarks, so the review cannot estimate one common effect size or claim that grounding always improves every metric.

H3 assessment. ChatAssert [3] provides evidence consistent with H3 by combining static analysis and dynamic execution feedback in an iterative repair loop and reporting improvement over its baseline. Cross-study comparisons remain descriptive because the baselines and evaluation datasets differ.

9.1 Why no pooled p-value is reported

This review does not have a harmonized dataset of per-test paired outcomes across all included studies. Reporting a new p-value from paper-level summary figures would violate statistical assumptions and risk false precision. Accordingly, hypothesis assessment here is evidence-based and descriptive; it is not presented as a new inferential experiment.

9.2 Recommended confirmatory experiment

To test the hypotheses rigorously, researchers should run the same test cases under controlled conditions: (A) one-shot generation using implementation context; (B) generation with static/dynamic feedback; and (C) generation grounded in independent documentation/specification. Use project-level train/test separation and post-cutoff or otherwise contamination-audited projects. Record compilation success, execution validity, false-positive rate on fixed code, bug exposure, and mutation score. Use paired comparisons on identical test cases. For binary outcomes such as compile success or bug exposed, use McNemar's test for paired observations when its assumptions are met. For paired continuous or ordinal outcomes such as mutation scores aggregated per project, use a paired t-test only when distributional assumptions are defensible; otherwise use the Wilcoxon signed-rank test. Report effect sizes and 95% confidence intervals, adjust for multiple comparisons (for example, Holm correction), and treat project as a clustering unit where appropriate. A mixed-effects model may be preferable when observations are nested within projects or models. No confirmatory test statistics are supplied in this paper because raw paired observations were not re-collected. The statistical plan is specified to make a future primary study reproducible rather than inventing results.

10. Major Findings

1. Reliability is multi-dimensional. Compilation and execution are necessary, but they do not independently prove that an oracle representsintended behaviour or detects meaningful faults.

2. Textual similarity is an incomplete quality measure. Equivalent assertions may differ in wording, while a similar-looking assertion may beweak or incorrect [2].

3. Static and dynamic feedback can help repair mechanically invalid candidates, but passing the current implementation does not guaranteesemantic correctness [3].

4. Documentation and specifications can provide evidence independent of implementation behaviour, provided those sources are accurate,current, and sufficiently precise [7], [11], [13].

5. More prompt context is not automatically better. Context relevance, behavioural authority, and contamination control are more importantthan simply increasing prompt length [8], [9]. 6

6. Benchmark realism affects measured performance. Multiple assertions, mixed oracle forms, and unknown insertion positions can revealweaknesses hidden by simplified tasks [10], [12].

7. Mutation testing and real-defect exposure provide stronger evidence of fault sensitivity than exact match alone, although each depends onthe benchmark and defect set.

8. Repair systems should report stage-wise outcomes. A candidate can fail at generation, compilation, execution, false-positive control, or faultdetection; one aggregate score can hide these failure points.

9. Study-reported percentages are not directly interchangeable. Differences in models, datasets, languages, baselines, and metric definitionslimit cross-study ranking and pooled statistical conclusions.

10. Human review remains important for high-impact software because automated checks cannot guarantee that the chosen behaviouralcontract matches stakeholder intent.

11. Recommendations

For industry and development teams

(i) Treat generated assertions as reviewable proposals, not authoritative specifications.
(ii) Run static checks, compilation, and test execution before accepting an oracle.
(iii) Where possible, check behaviour on both known-correct and faulty versions.
(iv) Use mutation testing to evaluate whether tests detect meaningful changes.
(v) Provide concise, accurate method contracts and documentation.
(vi) Record the model/version, prompt, source context, specification, repair history, and validation outcome for traceability.
(vii) Require human review for security-critical, financial, medical, or safety-related software.
(viii) Avoid assertions tied to incidental implementation details, unstable timing, or environment-specific values unless those are part of the contract.
(ix) Integrate generated oracles into CI as a gated workflow: compile, execute, check false positives, then evaluate fault sensitivity.
(x) Keep rollback and audit logs so a later-discovered weak oracle can be traced and corrected.

For researchers and benchmark authors

Publish benchmark versions, prompts, model identifiers, configuration details, scripts, and denominators. Report confidence intervals and effect sizes when raw data support them.

Separate exact match, compilation, execution validity, false-positive rate, mutation score, and real-bug exposure. Include failed generations and abstentions rather than reporting only successful cases.

Use developer-written tests, multiple oracle categories, project-level separation, and contamination audits. Describe the benchmark's limitations and avoid declaring a universal winner from a single metric.

12. Limitations of the Research

This is a focused review, not a fully exhaustive replication of searches across every database. Relevant studies may have been missed because of indexing, terminology, publication timing, or access limitations.

The synthesis relies on published descriptions and available artifacts; it does not rerun the primary experiments or independently reproduce every reported metric. Consequently, reported results are attributed to the original studies rather than presented as new findings.

Study designs differ in LLM versions, prompts, decoding settings, languages, benchmarks, baselines, and metric definitions. These differences prevent defensible pooling of effect sizes and make direct ranking uncertain.

Much of the empirical evidence is Java-centric, so conclusions may not transfer to Python, JavaScript, C/C++, integration tests, or industrial systems with

asynchronous and external dependencies.

Qualitative coding involves researcher judgment. Publication bias and model-version changes may also affect the available evidence and its reproducibility.

Mutation score depends on the selected mutation operators and surviving mutants; it is a useful proxy, not a complete measure of real-world defect detection. Human-written oracles are also imperfect and should not be assumed to be flawless ground truth.

13. Future Scope of Research

13.1 Standardized, contamination-aware benchmarks

Future work should create a shared benchmark combining developer-written tests, multiple assertions, exception checks, custom oracles, and runnable fixed/faulty program pairs. It should preserve realistic insertion positions, use project-level separation, document model versions and contamination audits, and publish scripts, prompts, environments, and expected outcomes.

13.2 Cross-language and cross-domain evaluation

Because much current evidence is Java-centric, studies should replicate methods in Python, JavaScript, C/C++, and other ecosystems. Results should be separated by language, project size, oracle type, and domain. Integration tests and libraries with asynchronous behaviour and external dependencies should also be included.

13.3 Stronger behavioural grounding

Compare implementation-only prompts with developer documentation, API contracts, formal specifications, and verified examples. Measure source completeness, freshness, ambiguity, and disagreement rather than assuming documentation is correct. When a specification conflicts with implementation behaviour, the system should flag the conflict rather than silently copy a potential defect.

13.4 Confidence and abstention

Future systems should estimate uncertainty and abstain or request developer review when specifications are incomplete, contradictory, or unreliable. Explanations should be evaluated for whether they help reviewers detect weak assertions, not merely whether they sound convincing.

13.5 Evaluation beyond accuracy

Report generation rate, compilation, execution on correct code, false-positive rate, real-defect exposure, mutation score, and residual error after each repair stage. Paired comparisons should use identical test cases, report effect sizes and confidence intervals, correct for multiple comparisons, and account for clustering by project where appropriate.

13.6 Practical deployment and human factors

Measure cost, latency, maintenance burden, and developer review time alongside reliability. User studies can examine acceptance, editing, and rejection rates and whether provenance logs help developers find defects. Critical software should retain human approval, audit logs, and rollback capability.

14. Conclusion

This focused systematic literature review examined recent methods for validating, repairing, grounding, and evaluating LLM-generated unit test oracles. The literature shows that a plausible and executable assertion can still misrepresent intended behaviour or fail to detect a meaningful defect. Oracle quality must therefore be evaluated through complementary checks rather than a single score.

Static analysis and execution feedback can improve mechanical validity; retrieval can supply useful examples; carefully selected context can help generation; and documentation or specifications can provide behavioural evidence independent of the implementation. Each intervention has limits: retrieved examples may not transfer, excess context can distract, documentation may be stale, and buggy code may mislead the model. Independent validation and traceability remain essential.

The studies also demonstrate why realistic benchmarks matter. Developer-written tests can contain multiple assertions and mixed oracle types, while simplified tasks may reveal insertion positions or reward textual resemblance. Because studies differ in models, datasets, and metrics, this review compares their findings descriptively and does not claim a pooled effect size or new statistical significance.

The hypotheses are supported descriptively, not confirmed by a new controlled experiment: the evidence indicates that textual similarity alone is inadequate, independent behavioural evidence may improve grounding, and feedback loops can improve mechanical validity. Confirmatory conclusions require reproducible experiments with paired observations, appropriate statistical tests, and confidence intervals.

Dependable LLM-assisted testing should be treated as a validation workflow, not one-shot text generation. Progress should be judged by whether an oracle accepts correct behaviour, rejects faulty behaviour, and provides measurable value with manageable cost and review effort. Standardized contamination-aware benchmarks, stage-wise reporting, human oversight, and reproducible artifacts are practical priorities for future research.

REFERENCES

1) E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," IEEE Transactionson Software Engineering, vol. 41, no. 5, pp. 507–525, 2015, doi: 10.1109/TSE.2014.2372785.

2) J. Shin, H. Hemmati, M. Wei, and S. Wang, "Assessing Evaluation Metrics for Neural Test Oracle Generation," IEEE Transactions on Software Engineering, vol. 50, no. 9, pp. 2337–2349, 2024, doi: 10.1109/TSE.2024.3433463.

3) I. Hayet, A. Scott, and M. d'Amorim, "ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance," IEEE Transactions on Software Engineering, vol. 51, no. 1, pp. 305–319, 2025, doi:10.1109/TSE.2024.3519159.

4) Q. Zhang et al., "Exploring Automated Assertion Generation via Large Language Models," ACM Transactions on Software Engineeringand Methodology, vol. 34, no. 3, 2025, doi: 10.1145/3699598.

5) S. B. Hossain and M. B. Dwyer, "TOGLL: Correct and Strong Test Oracle Generation with LLMs," in Proc. 47th IEEE/ACM International Conference on Software Engineering (ICSE), 2025, pp. 1475–1487, doi: 10.1109/ICSE55347.2025.00098.

6) S. M. Khandaker, F. M. Kifetew, D. Prandi, and A. Susi, "AugmenTest: Enhancing Tests with LLM-Driven Oracles," in Proc. IEEEInternational Conference on Software Testing, Verification and Validation (ICST), 2025, pp. 279–289, doi: 10.1109/ICST62969.2025.10988926.

7) S. B. Hossain, R. Taylor, and M. B. Dwyer, "Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation," Proceedings of the ACM on Software Engineering, vol. 2, FSE, Art. FSE084, pp. 1870–1891, 2025, doi: 10.1145/3729354.

8) D. Molinelli, L. Di Grazia, A. Martin-Lopez, M. D. Ernst, and

9) M. Pezzè, "Do LLMs Generate Useful Test Oracles? An Empirical Study with an 8 Unbiased Dataset," in Proc. 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2025, pp. 278–290, doi: 10.1109/ASE63991.2025.00031.

10) A. Bodicoat, G. Jahangirova, and V. Terragni, "Understanding LLM-Driven Test Oracle Generation," in Proc. ACM International Conference on AI-Powered Software (AIware), 2025, doi: 10.1109/AIware69974.2025.00011.

11) T. Tasnim, M. B. Dwyer, and S. B. Hossain, "TOGBench: A Developer-Written Multi-Variant Dataset and Benchmark Suite for TestOracle Generation," in Proc. 3rd ACM International Conference on AI-Powered Software (AIware), 2026, pp. 358–366, doi: 10.1145/3805760.3814927.

12) R. Fu, Y. Zhao, M. Wang, Z. Wang, and J. Chen, "LLM-Based Specification-Driven Test Oracle Enhancement," in Proc. 34th ACMInternational Conference on the Foundations of Software Engineering, Companion, 2026, pp. 1605–1613, doi: 10.1145/3803437.3805616.

13) H. Li, K. E, W. Sun, Q. Zhang, and M. Yan, "Towards More Realistic Assertion Generation under Mixed-Assertion Scenario," in Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2026.

14) J. Zhao, S. Zhou, and E. Cohen, "Evaluating and Mitigating the Misguidance Effect of Buggy Code in LLM-Generated Unit Tests,"Proceedings of the ACM on Software Engineering, ISSTA, 2026, doi: 10.1145/3832204.

15) B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," EBSE TechnicalReport EBSE-2007-01, 2007.

16) M. J. Page et al., "The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews," BMJ, vol. 372, Art. n71,2021, doi: 10.1136/bmj.n71.

ACKNOWLEDGMENT

The authors sincerely thank the Department of MCA, D Y Patil University School of Humanities and Sciences, for the academic support, facilities, and research environment provided during this work.