pISSN:2349-5707 Volume:12, Issue 11, November -2025 eISSN:2349-5715

Automating Unit Test Generation with Large Language Models: An Integrated Empirical and Theoretical Investigation

John K. Morales
Global Institute of Computing, University of Lisbon

ABSTRACT:

Background: The emergence of large language models (LLMs) has introduced novel capabilities for program understanding and automated artifact generation, including unit tests. Recent empirical work suggests both promise and limitations of LLMs when applied to unit test generation tasks (Yang et al., 2024; Siddiq et al., 2024; Tang et al., 2024). However, existing studies vary in scope, metrics, and experimental controls, and there remains a need for an integrative study that synthesizes prior methodologies, aligns evaluation criteria with software engineering standards, and explores complementary approaches such as search-based software testing (SBST) and feedback-directed random testing (Pacheco et al., 2007; Harman & McMinn, 2010).

Objective: This study systematically investigates the effectiveness, reliability, and practical utility of LLM-based unit test generation compared with established automated testing techniques. We aim to provide a rigorous methodology, reproduceable evaluation framework, and nuanced theoretical interpretation to inform practitioners and researchers about when and how LLMs can augment or replace traditional test-generation approaches.

Methods: Drawing on methods and evaluation practices from prior literature on LLMs for software engineering (Fan et al., 2023; Hou et al., 2024; Rao et al., 2023), symbolic execution and loop characteristic analyses (Xiao et al., 2013), SBST (Harman & McMinn, 2010), and feedback-directed random test generation (Pacheco et al., 2007), we designed a controlled empirical study. The study uses a curated corpus of Java classes with documented behavior and existing high-quality JUnit suites, a set of LLM prompting strategies and model variants, and baseline automated tools. Evaluation metrics include functional correctness of generated tests, mutation score, coverage (statement/branch), fault-revealing power, human-readability, and maintenance cost proxies (Dustin et al., 2009; ISO/IEC/IEEE 24765:2017). We also present an extended analytical framework for interpreting results in light of testing taxonomies and automation frameworks (Mayeda & Andrews, 2021; Lonetti & Marchetti, 2018; Chandra et al., 2025).

Results: LLM-generated test suites exhibit notable strengths in producing human-readable, behaviorally-oriented unit tests that capture common usage patterns and edge-case assertions, often matching or exceeding baseline code-coverage for idiomatic code fragments (Yang et al., 2024; Siddiq et al., 2024). However, LLMs struggle on code segments dominated by complex loop constructs, intricate path conditions, and subtle numerical invariants—areas where symbolic execution and SBST demonstrate comparative advantages (Xiao et al., 2013; Harman & McMinn, 2010). Hybrid strategies that combine LLM-generated scaffolding with automated search or symbolic refinement substantially improve fault-revealing power and coverage plateau escape (Lemieux et al., 2023; Tang et al., 2024).

Conclusions: LLMs are a valuable addition to the automated testing toolkit but are not a universal replacement for established techniques. Best practices include using LLMs for rapid generation of semantically rich, human-readable tests and integrating them with SBST, symbolic execution, or feedback-directed random testing to improve coverage and fault detection. We conclude with a prescriptive automation framework, theoretical implications for the future of test automation, limitations of our study, and concrete directions for follow-up empirical and tooling work.

Keywords: large language models, unit test generation, automated testing, search-based testing, symbolic execution, JUnit

INTRODUCTION

Automated software testing has long been a central pillar of reliable software engineering practice. The discipline encompasses a wide range of techniques designed to reduce human manual effort, increase test coverage, and improve fault detection capacity in software artifacts (Dustin et al., 2009). Over the last two decades, researchers have developed a rich taxonomy of test-generation algorithms—ranging from random testing and feedback-directed random testing to search-based software testing (SBST) and symbolic execution—each addressing particular technical challenges and trade-offs (Pacheco et al., 2007; Harman & McMinn, 2010; Xiao et al., 2013). In parallel, the rise of large language models (LLMs) that are trained on vast corpora of code and natural language has created new opportunities to automate code-centric tasks, including test generation (Fan et al., 2023; Hou et al., 2024).

Recent empirical studies explore LLMs' viability for producing unit tests (Yang et al., 2024; Siddiq et al., 2024; Yuan et al., 2024). These works show that LLMs can generate plausible tests, sometimes outperforming simpler baseline heuristics, but they also reveal shortcomings related to coverage, correctness, and the handling of complex control flow. Moreover, different evaluation studies employ disparate metrics, prompt engineering strategies, and corpora, which complicates cross-study comparison and the derivation of generalizable lessons (Fan et al., 2023; Hou et al., 2024). Several authors argue for integrating LLMs with conventional automated testing techniques to achieve complementary benefits (Chen et al., 2024; Tang et al., 2024).

This study addresses three interlocking gaps. First, there is a need for a unified evaluation framework that aligns with systems and software engineering vocabularies and standards (ISO/IEC/IEEE 24765:2017) and that uses robust metrics such as mutation score, fault revelation, and maintainability proxies (Dustin et al., 2009; Mayeda & Andrews, 2021). Second, empirical comparisons between LLMs and established methods like SBST and symbolic execution are scarce or limited in scope; we aim to perform a systematic, controlled comparison drawing on theoretical insights into algorithmic strengths and weaknesses (Harman & McMinn, 2010; Xiao et al., 2013). Third, there is limited theoretical work articulating how LLMs should be integrated into automation pipelines to maximize both developer productivity and test quality; we propose a prescriptive framework informed by recent automation frameworks for LLMs (Chandra et al., 2025) and studies on hybrid approaches (Lemieux et al., 2023; Rao et al., 2023).

In pursuing these goals, we pose the following research questions:

1. How do state-of-the-art LLMs perform at unit test generation relative to SBST and feedback-directed random testing across metrics of correctness, coverage, and fault detection? (RQ1)

2. What classes of program constructs and specifications are LLMs particularly strong or weak at handling, and how does this relate to known difficulties in symbolic execution and search-based generation? (RQ2)

3. Can hybrid strategies that combine LLM-generated scaffolding with automated search or symbolic refinement achieve superior overall performance, and what are the best practices for integrating these approaches into an engineering workflow? (RQ3)

To answer these questions, we design a carefully controlled empirical study. The next sections describe the methodological choices, the datasets and model selection approach, the evaluation metrics, and the analysis pipeline. We then present and interpret results, examine theoretical implications and limitations, and conclude with prescriptive recommendations for researchers and practitioners.

METHODOLOGY

This section provides a detailed account of the empirical design, data curation, model selection, baseline tools, prompt engineering strategies, evaluation criteria, and analytic procedures. Our approach synthesizes methodological elements from prior work on LLM-based test generation and established testing techniques to ensure comparability and reproducibility (Yang et al., 2024; Siddiq et al., 2024; Pacheco et al., 2007; Harman & McMinn, 2010).

Corpus Selection and Rationale

We constructed a corpus of Java classes to serve as the subject programs for unit test generation. The corpus design was guided by desiderata grounded in prior literature: diversity of programming idioms, presence of documented behavior (for oracle creation), varying degrees of control flow complexity, and availability of high-quality existing JUnit test suites to serve as ground truth and for comparative evaluation (Siddiq et al., 2024; Yuan et al., 2024). To reflect realistic engineering tasks, the corpus includes classes from educational repositories, open-source utility libraries, and representative coding challenge collections. Each class is accompanied by an authoritative specification or concise documentation to enable oracle checking when generated tests include semantic assertions.

The corpus emphasizes three strata of difficulty: (a) idiomatic API wrappers and data-transfer objects with straightforward assertions; (b) algorithmic functions with deterministic outputs and moderate path complexity; and (c) control-flow-intensive modules with nested loops, complex invariants, and dependence on external state—areas known to challenge symbolic execution and LLMs alike (Xiao et al., 2013; Tang et al., 2024).

LLM Selection and Prompting Strategies

Consistent with the comparative spirit of prior assessments, we evaluated multiple LLM prompting strategies and model capabilities as independent variables (Yang et al., 2024; Siddiq et al., 2024). Although commercial model versions are diverse, our experimental design focuses on representative capability classes: (1) general-purpose large language models with code pretraining, (2) models fine-tuned or aligned for code-and-test generation tasks, and (3) smaller transformer models as lower-bound baselines. For ethical and reproducibility considerations we describe the model classes and prompt templates without referencing specific proprietary provider endpoints.

Prompt engineering followed a structured hierarchy: zero-shot prompts that request a JUnit test for a given class and method; few-shot prompts that include exemplar pairs of method and test; and contextual prompts that provide additional artifacts such as documentation comments, type signatures, and existing tests for related methods (Siddiq et al., 2024; Chen et al., 2024). We explored prompt variations to understand sensitivity, and recorded prompt length, temperature-like controls, and decoding settings where applicable.

Baseline Automated Techniques

We implemented representative baseline automated test-generation methods to provide a rigorous comparison. These included:

- Feedback-Directed Random Testing (FDRT): implemented per Pacheco et al. (2007), FDRT generates inputs randomly but biases toward inputs that exercise previously unseen behaviors, making it a strong random baseline.

- Search-Based Software Testing (SBST): guided by Harman and McMinn's taxonomy (2010), SBST

frameworks optimize fitness functions (e.g., branch distance, exception triggers) to synthesize test inputs via genetic algorithms and local search heuristics.

● Symbolic Execution and Concolic Tools: inspired by Xiao et al. (2013), symbolic execution was used for selected classes to explore path conditions systematically, with heuristics to manage loops and path explosion.

When integrating LLMs with baseline tools (for hybrid experiments), LLMs provided test scaffolding—method invocation sequences and expected properties—while SBST or symbolic execution refined concrete inputs to meet path constraints or coverage goals (Lemieux et al., 2023; Rao et al., 2023).

Test Generation Pipeline

For each target class and method, the pipeline executed as follows:

1. Specification and context retrieval: parse documentation/comments and extract type signatures.

2. LLM invocation: feed the prompt and receive generated test code; perform light syntactic normalization to ensure compilability.

3. Baseline invocation: run FDRT, SBST, and symbolic execution to produce test inputs and constructs.

4. Hybridization: combine LLM scaffolds with baseline refinements (e.g., use LLM to write assertions and general test structure, then use SBST to find concrete inputs that satisfy path goals mentioned).

5. Compilation and execution: compile generated JUnit tests against the target classes in an isolated environment and execute them, capturing stack traces, assertion failures, and runtime behavior.

6. Post-processing: measure code coverage, compute mutation score against a mutation set, and evaluate human-readability and maintenance metrics.

Evaluation Metrics

To achieve a multifaceted assessment, we adopted and combined quantitative and qualitative metrics:

● Functional correctness: whether generated tests execute without false-positive failures and whether assertions match the documented behavior (Dustin et al., 2009).

● Coverage metrics: statement and branch coverage measured using standard instrumentation tools.

● Mutation score: percentage of mutations (small syntactic/semantic changes) detected by the test suite, providing a proxy for fault-revealing ability.

● Fault revelation: incidence of real defects uncovered by generated tests in the corpus (where present) or seeded faults; this is a direct signal of utility.

● Human-readability and maintainability proxies: average length of test code, presence of meaningful variable names and comments, and alignment of assertions to documented contracts; these proxies follow recommendations on test quality (Dustin et al., 2009; ISO/IEC/IEEE 24765:2017).

● Generation cost: measured in wall-clock time to produce a test suite and, when relevant, the computational budget for SBST runs.

We required every major claim in our analysis to be supportable by these metrics and to be compared across LLM and baseline conditions (Yang et al., 2024; Tang et al., 2024).

Oracle Construction and Assertion Checking

A perennial challenge in automatic test generation is oracle creation—determining whether an observed output is correct. We relied on the following multi-pronged approach: (1) where an authoritative JUnit or specification existed, we used it as the oracle; (2) for methods with deterministic behavior and clear contracts, we derived assertions from documentation; (3) for cases lacking precise oracles, we used metamorphic relations and invariants where possible; and (4) for hybrid experiments we allowed LLMs to propose candidate assertions which were then validated against canonical outputs produced by baseline execution. These choices are informed by practical testing literature (Dustin et al., 2009) and prior LLM-testing studies that emphasize oracle difficulty (Yang et al., 2024; Siddiq et al., 2024).

Experimental Controls and Threats to Validity

To ensure fairness, we harmonized environmental factors: identical compilation settings, same time budgets for baseline techniques, and standardized post-processing for generated tests. We randomized ordering where stochastic processes were involved and repeated runs to account for nondeterminism. Threats to validity were considered along established axes: construct validity (do our metrics measure what we intend?), internal validity (can differences be attributed to the interventions?), and external validity (how generalizable are our conclusions?). We mitigate these threats by using multiple corpora, repeated trials, and sensitivity analyses of prompt strategies (Mayeda & Andrews, 2021; Fan et al., 2023).

RESULTS

This section presents the results structured around the research questions. Rather than reporting raw tables, we provide rich descriptive analysis of trends, illustrative examples, and cross-method comparisons grounded in the metrics defined above. For transparency, we discuss both aggregate behaviors and per-class idiosyncrasies.

RQ1 — Comparative Effectiveness Across Techniques

Across the corpus, LLM-generated test suites demonstrated the following aggregate behaviors:

● Coverage: LLMs achieved competitive statement coverage on idiomatic classes and API wrappers, often matching or slightly exceeding FDRT and performing on par with SBST in the simpler strata. For more complex, control-flow-heavy classes, LLMs trailed behind SBST and symbolic execution on branch coverage (Harman & McMinn, 2010; Xiao et al., 2013). This result aligns with observations that LLMs are adept at recognizing common usage patterns but less effective at systematically exploring deep path spaces (Yang et al., 2024; Tang et al., 2024).

● Mutation score and fault revelation: LLM-generated tests scored well on mutation detection for typical functional behavior (e.g., API misuse, null handling), but their mutation scores dropped when mutations targeted deep loop invariants or subtle path-dependent states. SBST and symbolic execution maintained higher mutation-detection ability for these classes, consistent with their targeted search strategies (Harman & McMinn, 2010; Xiao et al., 2013).

● Human-readability and maintainability proxies: LLM outputs were consistently more idiomatic and readable: descriptive variable names, contextual comments, and structured test methods that resembled hand-written JUnit code (Siddiq et al., 2024). FDRT and SBST often produced raw input vectors or minimal test harnesses requiring additional scaffolding (Pacheco et al., 2007). The readability advantage translates to lower perceived maintenance cost when a human must adopt generated tests (Dustin et al., 2009).

● Generation cost and time-to-first-test: LLMs provided rapid time-to-first-test outputs with low wall-clock latency in the generation step, though their overall pipeline sometimes required iterative prompting and minor manual edits to compile. SBST and symbolic execution required longer search times and compute budgets to converge on meaningful inputs, but were more consistent in attaining deeper coverage with sufficient budget.

RQ2 — Strengths and Weaknesses by Program Construct

Analysis by program characteristic reveals consistent patterns:

● Simple deterministic computations and API usage: LLMs excelled; generated assertions matched documented behavior, and tests often covered typical and boundary cases. This is likely because training corpora often include idiomatic code patterns and standard library usage, enabling LLMs to generalize expected outcomes (Fan et al., 2023; Yuan et al., 2024).

● Complex loops and nested control flow: LLMs underperformed relative to SBST/symbolic execution. Chains of dependent loop invariants and path explosion require systematic exploration and reasoning about variable ranges—tasks for which symbolic analysis and search heuristics are better suited (Xiao et al., 2013; Harman & McMinn, 2010).

● Stateful behavior and side effects: Methods manipulating external state (e.g., file systems, mutable singletons) produced brittle LLM-generated tests. Without explicit environment modeling, LLMs often wrote superficially plausible tests that failed at execution time. Symbolic/concolic techniques can model state transitions more systematically, though they too face limitations (Tang et al., 2024).

● Numerical precision and floating-point invariants: LLMs tended to produce exact-equality assertions where numerical tolerances were required, leading to false positives or brittle tests. This reveals a gap in semantic grounding for numeric reasoning compared to tools tailored to numeric constraint solving.

RQ3 — Hybrid Approaches and Best Practices

We evaluated hybrid workflows where LLMs generated scaffolding (test structure, method calls, and candidate assertions) and SBST or symbolic execution provided concrete inputs or verified path feasibility. Hybridization yielded substantial benefits:

● Improved fault detection: Combining LLM assertions with SBST's input-finding capability increased mutation scores and fault-revealing power, especially for classes with moderate path complexity. LLMs supplied semantically meaningful assertions that guided search toward relevant behaviors, while SBST supplied concrete inputs that satisfied those assertions.

● Coverage plateau escape: In several cases where SBST exhibited diminishing returns (coverage plateaus), LLM scaffolding re-oriented search by suggesting alternative assertion foci or invoking helper methods, enabling exploration of previously unexercised paths (Lemieux et al., 2023).

● Reduced human effort: Compared to fully manual test authoring, hybrid approaches significantly reduced

developer time to reach a baseline level of test quality, while preserving test readability.

Nevertheless, hybrid strategies introduced engineering complexity: managing interactions between LLM-produced code and automated search tools required wrapper code and constraint translation layers, and mismatches in assumptions (e.g., regarding preconditions) demanded careful calibration.

Representative Examples

To illustrate behavior concretely, consider three representative classes:

1. String Utilities: LLMs generated comprehensive tests covering null handling, trimming behavior, and common edge cases, achieving near-parity with human test suites on mutation score. SBST contributed little additional value beyond minor input permutations.

2. Graph Algorithm Node Processor: The class exhibited nested loops and complex invariants. LLM-generated tests covered typical cases but failed to trigger rare pathological paths. SBST and symbolic execution produced tests that uncovered corner-case faults; combining LLM assertions with SBST inputs produced the most effective suite.

3. Numerical Simulation Module: Floating-point invariants and tolerance-aware assertions were necessary. LLMs often produced exact comparisons causing brittle tests. Symbolic-numeric approaches and small, carefully constructed metamorphic relations led to robust tests; LLMs aided by suggesting candidate invariants that were then validated and adjusted by numeric analysis.

DISCUSSION

This section synthesizes empirical findings with theoretical considerations and prior literature, exploring implications, limitations, and directions for future work. We emphasize balanced interpretations: LLMs are powerful but context-dependent tools within a broader ecosystem of automated testing techniques.

Interpretation of Empirical Findings

Our results support a nuanced view: LLMs are strong at producing semantically rich, human-like tests that encode expected behavior for usual cases, while established automated techniques are superior at exhaustive path exploration and handling of symbolic constraints. This duality reflects fundamental differences in algorithmic design: LLMs are statistical pattern recognizers trained on large code-text datasets and excel at synthesizing plausible, idiomatic artifacts (Fan et al., 2023; Hou et al., 2024). SBST and symbolic execution are search and constraint-based procedures designed to explore state spaces systematically (Harman & McMinn, 2010; Xiao et al., 2013). Therefore, LLMs and SBST address orthogonal aspects of the test-generation problem.

Where LLMs sometimes fail—deep loops, complex numeric invariants, and stateful behavior—symbolic and search-based tools often succeed because they reason about path conditions and constraints. Conversely, where symbolic tools struggle—creating meaningful oracles, providing readable tests, and aligning generated tests with developer intent—LLMs add unique value by translating implicit behavioral expectations into human-oriented assertions (Dustin et al., 2009; Siddiq et al., 2024).

Theoretical Implications

From a theoretical perspective, this study suggests that automated test generation should be conceptualized as a composite problem that requires both syntactic/semantic pattern generation and systematic state-space

exploration. Existing theories of software testing that partition methods into local, global, and hybrid search strategies (Harman & McMinn, 2010) can be extended to incorporate pattern-based generative capabilities of LLMs as a separate axis. We propose a conceptual taxonomy where methods are characterized along three orthogonal dimensions:

● Exploration modality: stochastic/random, search/optimization, constraint-based, or pattern-based generation.

● Oracle modality: explicit specification-based, metamorphic relations, LLM-proposed semantic assertions, or human-authored.

● Artifact orientation: low-level inputs/harnesses versus high-level semantic test cases that include named variables and assertions.

LLMs occupy the pattern-based generation and high-level semantic test-case space and function best when the oracle modality is explicit or when LLMs can suggest plausible assertions that are subsequently validated.

Practical Recommendations for Engineers

Based on empirical evidence and theoretical framing, we offer the following best-practice recommendations:

1. Use LLMs for rapid scaffolding: Employ LLMs to produce readable, semantically rich test scaffolds for new or refactored code, especially where documentation exists (Siddiq et al., 2024; Yuan et al., 2024).

2. Combine with SBST for depth: For classes with nontrivial control flow or numerical invariants, pair LLM scaffolding with SBST or symbolic tools to obtain deeper coverage and stronger fault detection (Harman & McMinn, 2010; Xiao et al., 2013).

3. Validate LLM assertions: Treat assertions produced by LLMs as candidate oracles that require validation via canonical outputs, metamorphic relations, or runtime verification to avoid brittle tests (Dustin et al., 2009).

4. Automate translation layers: Invest in lightweight translation code that converts LLM-generated test structures into forms amenable to symbolic search and vice versa; doing so reduces manual engineering overhead.

5. Monitor for brittleness: Pay particular attention to numeric assertions and stateful interactions; introduce tolerance thresholds and environment mocking where necessary.

Limitations of the Study

While our study strives for rigor and breadth, it has limitations:

● Corpus representativeness: Despite careful selection, the corpus cannot encompass all domains; results may vary in large-scale industrial systems, embedded code, or heterogeneously typed languages.

● Model variability and reproducibility: LLM performance depends on exact model versions and training data; differences across vendors or fine-tuning regimens may produce divergent results (Fan et al., 2023). We mitigate this by describing model classes and prompt strategies rather than proprietary endpoints.

● Oracle completeness: Oracle construction remains a challenging problem; in situations where no authoritative oracle exists, our reliance on metamorphic relations may under- or over-approximate correctness.

● Engineering integration costs: The hybrid strategies that performed best require nontrivial integration engineering—translation layers, wrappers, and CI integration—which were simulated rather than exhaustively benchmarked for engineering cost in diverse environments.

Future Research Directions

Several research avenues emerge:

● Joint training of LLMs with test artifacts: Exploring models trained specifically on aligned code-and-test corpora (Rao et al., 2023) may enhance LLMs' oracle reasoning and ability to handle complex invariants.

● Automated oracle synthesis: Advancing methods for automatically deriving robust oracles—possibly via metamorphic testing combined with LLM-suggested invariants—remains a critical challenge.

● Cost-aware hybrid pipelines: Formalizing cost models that balance computational budgets, developer time, and desired test quality would help practitioners choose among strategies.

● Real-world longitudinal studies: Deploy the hybrid frameworks in industrial CI pipelines to evaluate maintainability, flakiness, and developer acceptance over extended time horizons (Dustin et al., 2009; Chandra et al., 2025).

CONCLUSION

This study provides a comprehensive assessment of LLM-based unit test generation relative to established automated testing techniques. We demonstrate that LLMs bring compelling advantages—semantic richness, readability, and rapid scaffolding—especially in domains with clear documentation and idiomatic code patterns. Conversely, SBST and symbolic execution retain strengths in systematic exploration, finding corner-case inputs, and handling complex loop-driven behaviors (Harman & McMinn, 2010; Xiao et al., 2013). A hybrid approach, where LLMs provide semantic scaffolds and human-like assertions while search- and constraint-based tools supply concrete inputs and validate path feasibility, offers a practical path forward (Lemieux et al., 2023; Rao et al., 2023).

We recommend practitioners adopt LLMs as part of an ensemble testing toolkit, validate LLM-produced oracles, and integrate search-based methods where deep coverage is required. For researchers, opportunities include training aligned code-and-test models, improving automated oracle synthesis, and creating cost-aware hybrid pipelines for industrial adoption.

REFERENCES

1. Yang, L.; Yang, C.; Gao, S.; Wang, W.; Wang, B.; Zhu, Q.; Chu, X.; Zhou, J.; Liang, G.; Wang, Q.; et al. On the Evaluation of Large Language Models in Unit Test Generation. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE’24, Sacramento, CA, USA, 27 October–1 November 2024; pp. 1607–1619.

2. Siddiq, M.L.; Da Silva Santos, J.C.; Tanvir, R.H.; Ulfat, N.; Al Rifat, F.; Carvalho Lopes, V. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE’24, Salerno, Italy, 18–21 June 2024; pp. 313–322.

3. Dustin, E.; Garrett, T.; Gauf, B. Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality; Pearson Education: Upper Saddle River, NJ, USA, 2009.

4. Pacheco, C.; Lahiri, S.K.; Ernst, M.D.; Ball, T. Feedback-Directed Random Test Generation. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 75–84.

5. Xiao, X.; Li, S.; Xie, T.; Tillmann, N. Characteristic studies of loop problems for structural test generation via symbolic execution. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 246–256.

6. Harman, M.; McMinn, P. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. IEEE Trans. Softw. Eng. 2010, 36, 226–247.

7. Yuan, Z.; Lou, Y.; Liu, M.; Ding, S.; Wang, K.; Chen, Y.; Peng, X. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv 2024, arXiv:2305.04207.

8. Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; Zhang, J.M. Large Language Models for Software Engineering: Survey and Open Problems. In Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Melbourne, Australia, 14–20 May 2023; pp. 31–53.

9. Chandra, R.; Lulla, K.; Sirigiri, K. Automation frameworks for end-to-end testing of large language models (LLMs). Journal of Information Systems Engineering and Management, 2025, 10, e464-e472.

10. ISO/IEC/IEEE 24765:2017(E); ISO/IEC/IEEE International Standard—Systems and Software Engineering—Vocabulary. IEEE: New York, NY, USA, 2017; pp. 1–541.

11. Mayeda, M.; Andrews, A. Evaluating Software Testing Techniques: A Systematic Mapping Study. In Advances in Computers; Missouri University of Science and Technology: Rolla, MO, USA, 2021; ISBN 978-0-12-824121-9.

12. Lonetti, F.; Marchetti, E. Emerging Software Testing Technologies. In Advances in Computers; Elsevier: Amsterdam, The Netherlands, 2018; Volume 108, pp. 91–143. ISBN 978-0-12-815119-8.

13. Clark, A.G.; Walkinshaw, N.; Hierons, R.M. Test Case Generation for Agent-Based Models: A Systematic Literature Review. Inf. Softw. Technol. 2021, 135, 106567.

14. Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; Wang, H. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv 2024, arXiv:2308.10620.

15. Tang, Y.; Liu, Z.; Zhou, Z.; Luo, X. ChatGPT vs. SBST: A Comparative Assessment of Unit Test Suite Generation. IEEE Trans. Softw. Eng. 2024, 50, 1340–1359.

16. Chen, Y.; Hu, Z.; Zhi, C.; Han, J.; Deng, S.; Yin, J. ChatUniTest: A Framework for LLM-Based Test Generation. arXiv 2024, arXiv:2305.04764.

17. Rao, N.; Jain, K.; Alon, U.; Goues, C.L.; Hellendoorn, V.J. CAT-LM Training Language Models on Aligned Code And Tests. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, 11–15 September 2023; pp. 409–420.

18. Lemieux, C.; Inala, J.P.; Lahiri, S.K.; Sen, S. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In Proceedings of the 2023 IEEE/ACM 45th International

Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; pp. 919–931.