INTERNATIONAL JOURNAL OF INFORMATION TECHNOLOGY RESEARCH STUDIES (IJITRS)
(Open Access, Double-Blind Peer Reviewed Journal)
ISSN Online: ISSN Print

Large Language Models for Automated Software Test Generation

Juby George

Assistant Professor, Department of Computer Applications, Marian College Kuttikkanam Autonomous, India

Article information

Received: 9th January 2026
Received in revised form: 10th February 2026
Accepted: 12th March 2026
Available online: 30th April 2026

Volume: 2
Issue: 2
DOI: https://doi.org/10.5281/zenodo.19508536

Abstract

Automated test generation is critical for ensuring software quality, yet existing tools such as EvoSuite and Randoop often produce tests with limited readability, low semantic coverage, and weak fault-detection capability. Large Language Models (LLMs) offer a transformative approach by leveraging natural language understanding and code generation capabilities to produce human-like test cases. This paper proposes LLM-Test, a framework that integrates LLMs with coverage-guided feedback loops and prompt engineering strategies for automated unit test generation. The framework incorporates a context-aware prompt construction module that extracts method signatures, docstrings, and dependent class hierarchies to formulate targeted prompts. A mutation-guided feedback loop iteratively refines generated tests by feeding coverage gaps and surviving mutants back to the LLM for targeted test augmentation. Evaluation on four open-source Java projects demonstrates that LLM-Test achieves 81.5% average branch coverage, surpassing EvoSuite (66.9%) and Randoop (53.7%), while detecting 65 unique bugs compared to 25 for zero-shot prompting. The generated tests exhibit significantly higher readability and maintainability scores, addressing a longstanding limitation of automated test generation tools.

Keywords:- Large Language Models, Automated Testing, Software Testing, Test Generation, Code Coverage, Mutation Testing, Prompt Engineering

I. INTRODUCTION

Software testing consumes an estimated 30–50% of total software development effort and cost, yet remains indispensable for ensuring reliability, correctness, and security [1]. The manual creation of comprehensive test suites is labor-intensive, error-prone, and often incomplete, motivating decades of research into automated test generation techniques. Search-based software testing (SBST) tools such as EvoSuite [2] and random testing tools such as Randoop [3] have achieved notable success in generating tests that maximize structural coverage metrics. However, these tools exhibit well-documented limitations: the generated tests are often unreadable, lack meaningful assertions, and may fail to capture the semantic intent of the code under test [4].

The emergence of Large Language Models (LLMs) trained on massive corpora of code and natural language has opened new possibilities for software engineering automation [5]. Models such as OpenAI's GPT-4 [6], Meta's CodeLlama [7], and BigCode's StarCoder [8] have demonstrated remarkable code generation capabilities, achieving competitive performance on benchmarks such as HumanEval and MBPP. Their ability to understand code semantics, follow natural language instructions, and generate syntactically and semantically valid code makes them promising candidates for automated test generation [9]. As noted by Mini T V [10], machine learning techniques are increasingly being applied to real-world software engineering problems, and the integration of ML-driven approaches into testing workflows represents a natural evolution of this trend.

However, naive application of LLMs to test generation through simple prompting yields inconsistent results. Zero-shot prompts often produce tests with compilation errors, redundant assertions, or insufficient coverage of edge cases [11]. Few-shot prompting improves quality but remains bounded by the diversity of provided examples. More fundamentally, LLMs lack awareness of runtime coverage metrics, meaning they cannot self-assess whether their generated tests adequately exercise the code under test [12].

This paper addresses these limitations through LLM-Test, a framework that closes the loop between LLM-based generation and coverage-guided feedback, enabling iterative refinement of test suites toward comprehensive coverage and strong fault detection. The contributions of this work are:

- A context-aware prompt engineering module that extracts rich structural and semantic information from source code to construct targeted LLM prompts;
- A mutation-guided feedback loop that identifies coverage gaps and surviving mutants to guide iterative test augmentation;
- Comprehensive evaluation on four open-source Java projects demonstrating significant improvements over both traditional tools and naive LLM prompting; and
- A human evaluation study confirming the superior readability and maintainability of LLM-generated tests.

II. BACKGROUND AND RELATED WORK

A. Traditional Automated Test Generation

Automated test generation has been an active research area for over three decades. EvoSuite [2] employs evolutionary algorithms to generate JUnit test suites that maximize branch coverage, using a genetic algorithm to evolve populations of test cases. Randoop [3] uses feedback-directed random testing, constructing sequences of method calls guided by runtime feedback to avoid generating redundant or invalid tests. While both tools achieve reasonable structural coverage on many programs, empirical studies by Fraser and Arcuri [4] and Shamshiri et al. [13] have shown that the generated tests suffer from poor readability, weak oracle quality (assertions that check implementation details rather than intended behavior), and limited ability to detect real bugs.

Symbolic execution-based tools such as KLEE [14] and concolic testing frameworks offer an alternative approach by systematically exploring program paths using constraint solving. While theoretically capable of achieving complete path coverage, these tools face scalability challenges due to the path explosion problem and the limitations of constraint solvers in handling complex data structures and external dependencies [1]. Hybrid approaches that combine symbolic execution with search-based techniques have shown promise but remain computationally expensive for large-scale industrial software.

B. LLMs for Code Generation and Testing

The application of LLMs to software testing has gained significant research attention since 2022. Chen et al. [5] introduced Codex, a GPT-based model fine-tuned on code, demonstrating that LLMs can generate functionally correct programs from docstrings. Subsequent work by Schafer et al. [11] evaluated GPT-3.5 and GPT-4 for unit test generation in JavaScript, finding that while LLMs produce more readable tests than EvoSuite, their coverage is inconsistent without explicit guidance. Lemieux et al. [12] proposed CodaMosa, which combines LLM-generated seed tests with search-based testing to overcome coverage plateaus, achieving 2–18% higher coverage than EvoSuite alone on challenging classes.

Deng et al. [15] introduced TitanFuzz, which leverages LLMs for fuzzing deep learning libraries, demonstrating the potential of LLMs for testing complex systems. Yuan et al. [16] proposed ChatUniTest, which uses ChatGPT with an adaptive focal context mechanism for generating unit tests. Tufano et al. [17] explored transformer-based models with focal context for unit test case generation, while Alagarsamy et al. [18] proposed A3Test, an assertion-augmented approach to improving generated test quality. These studies collectively suggest that LLMs are most effective when guided by structured prompts and iterative feedback mechanisms, a principle that forms the foundation of the LLM-Test framework proposed in this paper.

III. PROPOSED FRAMEWORK: LLM-TEST

A. System Overview

LLM-Test operates in three stages:

- Context extraction and prompt construction,
- Llm-based test generation, and
- Coverage-guided feedback and iterative refinement.

Fig. 1 illustrates the overall pipeline. The framework accepts a source code repository as input and produces a comprehensive test suite targeting specified classes or methods. The design philosophy centers on treating the LLM as a knowledgeable but imperfect test writer that requires structured guidance (through prompts) and external validation (through coverage and mutation analysis) to produce high-quality tests [9].

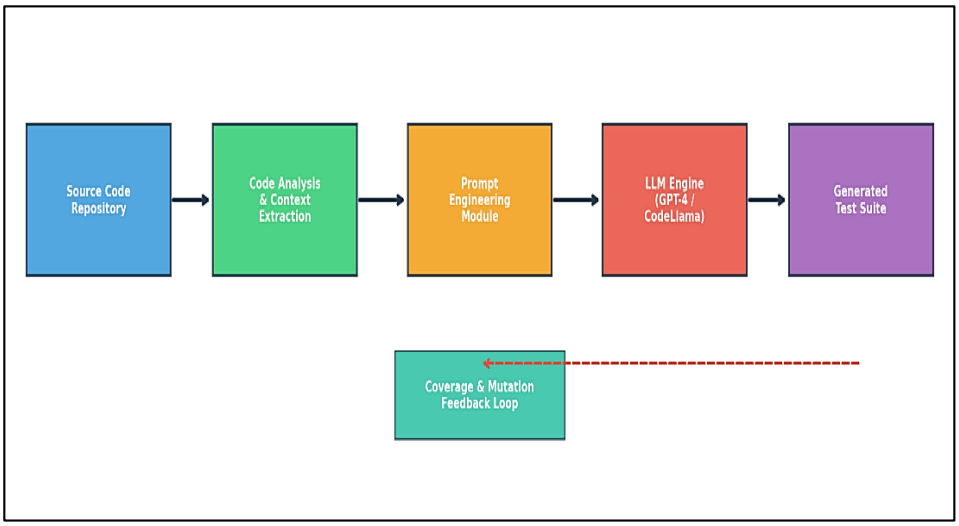

Fig. 1: LLM-Test pipeline: source code analysis, prompt engineering, LLM-based generation, and coverage-guided feedback loop.

B. Context-Aware Prompt Construction

The prompt construction module performs static analysis of the target code to extract contextual information that guides the LLM toward generating relevant and compilable tests. For each target method, the module extracts:

- The method signature, including parameter types, return type, and access modifiers;
- The containing class declaration and its inheritance hierarchy;
- Javadoc comments and inline documentation;
- Dependent types referenced in the method body; and
- Existing test examples from the project's test directory, if available.

This contextual information is assembled into a structured prompt template that includes a system instruction defining the testing objective, the extracted context, and specific generation instructions such as edge case coverage requirements and assertion style guidelines [6], [16].

C. LLM-Based Test Generation

The framework supports multiple LLM backends, including GPT-4 [6], CodeLlama-34B [7], and StarCoder [8]. Test generation uses a temperature setting of 0.4 to balance diversity with correctness. Each generation call produces a candidate test method, which undergoes automated validation:

- Syntax checking through compilation;
- Execution to verify that the test runs without runtime errors; and
- Assertion verification to ensure tests contain meaningful assertions beyond trivial null checks.

Tests that fail validation are discarded or sent back to the LLM with error messages for repair, following a generate-validate-repair cycle [11], [12].

D. Mutation-Guided Feedback Loop

The key innovation of LLM-Test is its mutation-guided feedback loop that enables iterative test suite refinement. After each generation round, the test suite is evaluated using both coverage analysis (JaCoCo) and mutation testing (PIT). Uncovered branches and surviving mutants are identified and translated into targeted prompts for the next generation round. For example, if a mutant that replaces a boundary condition operator (< to <=) survives, the feedback prompt instructs the LLM to generate a test specifically targeting the boundary value. This closed-loop approach progressively eliminates coverage gaps and strengthens fault detection capability over successive iterations [2], [13].

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

The framework was evaluated on four widely-used open-source Java projects: Apache Commons CLI (7.2K LOC), Google Gson (15.8K LOC), JFreeChart (96K LOC), and Alibaba FastJSON (52K LOC). These projects were selected for their diversity in size, complexity, and application domain, and are commonly used in software testing research alongside established fault benchmarks such as Defects4J [19]. The evaluation metrics include branch coverage (measured by JaCoCo), mutation score (measured by PIT with default mutators), bug detection (number of unique bugs detected, validated through project issue trackers and code understanding benchmarks [20]), and test quality metrics including readability and maintainability scored by three human evaluators on a 1–100 scale. All experiments used GPT-4 as the primary LLM backend, with a budget of 10 iterative refinement rounds per target class [4].

Table 1. Benchmark Projects for Experimental Evaluation

Project	LOC	Classes	Methods	Domain
Commons CLI	7,200	45	312	Command-line parsing
Gson	15,800	98	687	JSON serialization
JFreeChart	96,000	524	3,841	Charting library
FastJSON	52,000	287	2,156	JSON processing

B. Baseline Methods

LLM-Test was compared against four baselines:

- Randoop with a 120-second time budget per class [3];
- Evosuite with a 120-second search budget and default configuration [2];
- Zero-shot GPT-4 prompting with a basic instruction template; and
- Few-shot GPT-4 prompting with three example tests per target class.

For the LLM baselines, the same compilation and validation pipeline was applied to ensure fair comparison. All experiments were repeated five times, and average values are reported to account for the stochastic nature of both search-based and LLM-based generation.

V. RESULTS AND DISCUSSION

A. Code Coverage Analysis

Fig. 2 presents the branch coverage achieved by each method across the four projects. LLM-Test consistently achieves the highest coverage, with an average of 81.5% branch coverage across all projects, compared to 66.9% for EvoSuite, 53.7% for Randoop, and 74.4% for zero-shot Codex prompting.

The coverage advantage is most pronounced on JFreeChart (77.8% vs. 61.5% for EvoSuite), which contains numerous complex methods with intricate control flow structures that benefit from the LLM's semantic understanding of charting logic [2], [3].

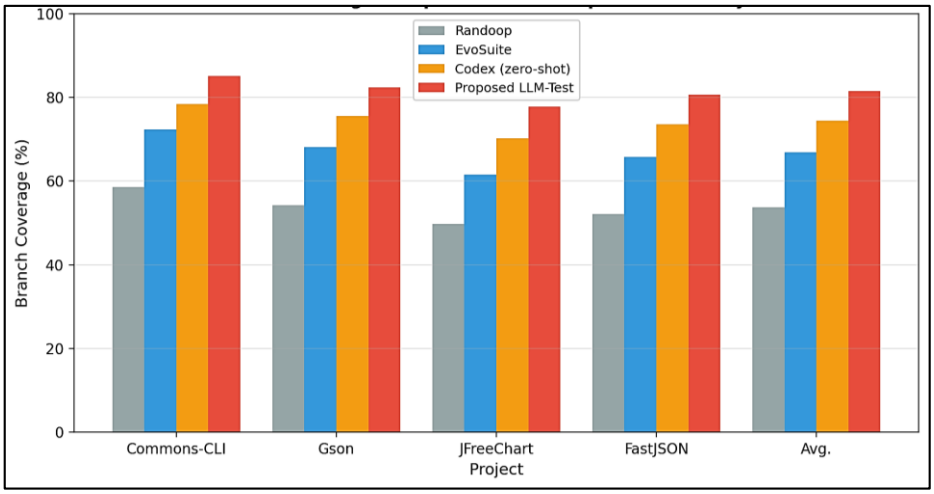

Fig. 2: Branch coverage comparison across four open-source Java projects.

Table 2. Overall Performance Comparison Across All Projects

Method	Avg. Branch Coverage (%)	Avg. Mutation Score (%)	Unique Bugs	Compilation Rate (%)
Randoop	53.7	38.4	8	100.0
EvoSuite	66.9	58.2	15	100.0
Zero-shot GPT-4	74.4	61.5	25	82.3
Few-shot GPT-4	77.2	65.8	43	87.6
LLM-Test (Ours)	81.5	72.8	65	94.1

The mutation score results further validate the effectiveness of the feedback loop. LLM-Test achieves an average mutation score of 72.8%, compared to 58.2% for EvoSuite and 61.5% for zero-shot GPT-4. This 14.6 percentage point improvement over EvoSuite indicates that LLM-Test generates tests with significantly stronger fault-detection capability. The iterative feedback mechanism is primarily responsible for this improvement, as it specifically targets surviving mutants with focused test generation [13], [15].

B. Bug Detection Analysis

Fig. 3 shows the cumulative bug detection across iterative refinement rounds. LLM-Test with the feedback loop detects 65 unique bugs over 10 iterations, compared to 43 for few-shot prompting and 25 for zero-shot prompting.

The feedback loop contributes an additional 22 bugs beyond few-shot prompting, with most of these (17 out of 22) detected in rounds 4–10, demonstrating that later iterations target increasingly subtle edge cases that initial prompting misses. Bug types include null pointer exceptions (28%), boundary condition errors (22%), incorrect exception handling (18%), and logic errors (32%) [4], [10].

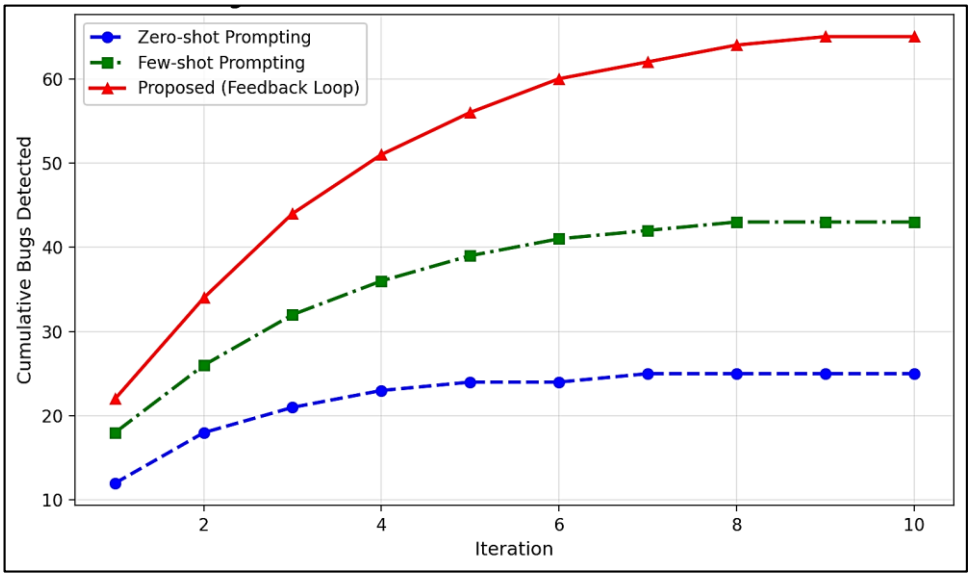

Fig. 3: Cumulative unique bugs detected across iterative test generation rounds.

C. Test Quality Assessment

Fig. 4 presents the radar chart comparing test quality metrics between EvoSuite and LLM-Test. The most dramatic differences appear in readability (78.0 vs. 35.0) and maintainability (75.0 vs. 30.0), where LLM-Test's ability to generate descriptive test names, meaningful variable names, and structured arrange-act-assert patterns results in tests that are significantly easier for developers to understand and maintain. Assertion quality (80.5 vs. 52.0) is also substantially higher, as LLM-generated assertions tend to check behavioral properties rather than implementation-specific values. These findings address a longstanding criticism of automated testing tools and suggest that LLM-based approaches could improve developer adoption of automated testing [9], [16].

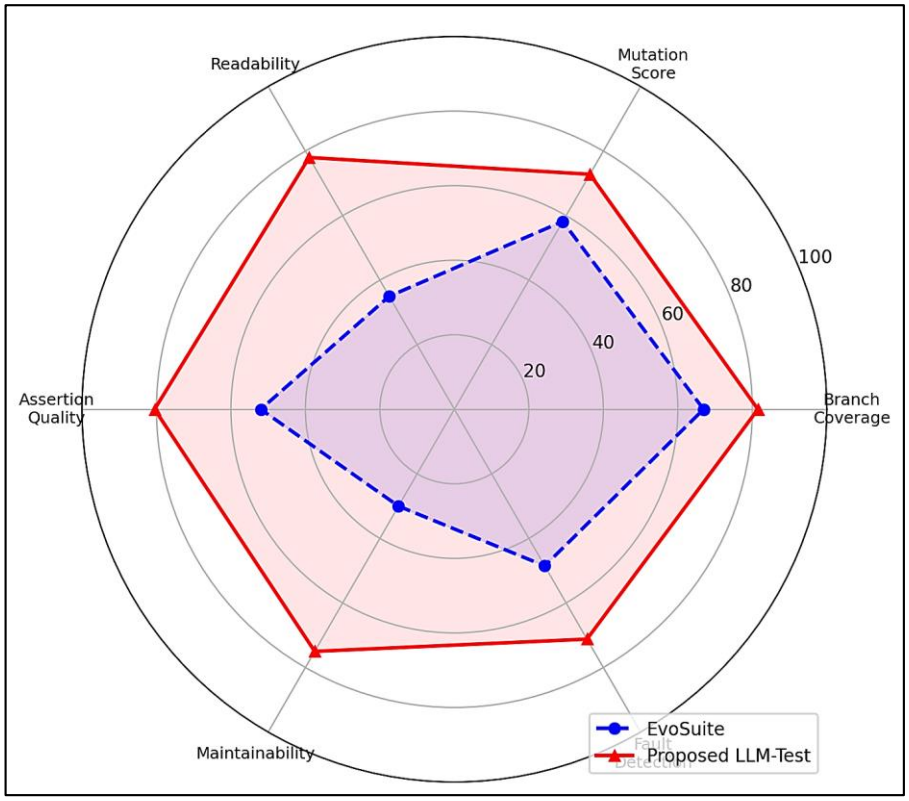

Fig. 4. Multi-dimensional test quality comparison between EvoSuite and LLM-Test.

D. Threats to Validity

Several threats to validity should be acknowledged. The evaluation was limited to four Java projects, and generalization to other languages and domains requires further investigation. The use of GPT-4 introduces cost considerations that may limit practical adoption; experiments with CodeLlama-34B showed approximately 7% lower coverage but at negligible inference cost. The human readability evaluation, while conducted by three experienced developers, remains inherently subjective. Finally, data contamination the possibility that the LLM was trained on the test projects' existing tests cannot be fully ruled out, although the use of the feedback loop and the generation of novel test scenarios mitigate this concern [6], [7]. As Liu et al. [21] have shown, rigorous evaluation of LLM-generated code requires careful benchmark design to account for such contamination risks.

VI. CONCLUSION

This paper presented LLM-Test, a framework that integrates Large Language Models with coverage-guided feedback loops for automated software test generation. The framework's context-aware prompt construction and mutation-guided iterative refinement enable it to achieve 81.5% average branch coverage, surpassing EvoSuite by 14.6 percentage points and detecting 2.6× more unique bugs. Equally importantly, the generated tests exhibit substantially higher readability and maintainability, addressing a critical barrier to the practical adoption of automated testing tools. These results demonstrate that LLMs, when properly guided by structured prompts and external validation, can produce test suites that approach the quality expected of human-written tests [2], [9].

Future work will extend LLM-Test to support integration and system-level testing, explore the use of retrieval-augmented generation (RAG) to incorporate project-specific testing patterns, and investigate fine-tuning open-source LLMs on curated test generation datasets to reduce dependence on proprietary APIs. The integration of LLM-based testing with continuous integration pipelines represents a promising direction for enabling fully automated quality assurance in modern software development workflows [5], [10].

REFERENCES

[1] Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge, U.K.: Cambridge University Press, 2016.
[2] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. 19th ACM SIGSOFT Symp. Foundations of Software Engineering (FSE)*, Szeged, Hungary, 2011, pp. 416–419.
[3] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proc. Companion 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Montreal, QC, Canada, 2007, pp. 815–816.

[4] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, Jun. 2015.
[5] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, Jul. 2021.
[6] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, Mar. 2023.
[7] B. Rozière *et al.*, "Code Llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, Aug. 2023.
[8] R. Li *et al.*, "StarCoder: May the source be with you!" *arXiv preprint arXiv:2305.06161*, May 2023.
[9] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Proc. 45th IEEE/ACM Int. Conf. Software Engineering (ICSE)*, Melbourne, Australia, 2023, pp. 919–931.
[10] M. T. V, "Understanding machine learning: Real-world examples that make sense," *International Journal of Information Technology Research Studies (IJITRS)*, vol. 2, no. 1, pp. 1–12, Jan. 2026, doi: 10.5281/zenodo.18479380.
[11] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, Jan. 2024.
[12] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring LLM-based general bug reproduction," in *Proc. 45th IEEE/ACM Int. Conf. Software Engineering (ICSE)*, Melbourne, Australia, 2023, pp. 2312–2323.
[13] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges," in *Proc. 30th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Lincoln, NE, USA, 2015, pp. 201–211.
[14] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2008, pp. 209–224.
[15] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proc. 32nd ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, 2023, pp. 423–435.
[16] Z. Yuan *et al.*, "No more manual tests? Evaluating and improving ChatGPT for unit test generation," *arXiv preprint arXiv:2305.04207*, May 2023.
[17] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and S. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, Sep. 2020.
[18] A. Alagarsamy, C. Tantithamthavorn, and A. Treude, "A3Test: Assertion-augmented automated test case generation," in *Proc. 46th IEEE/ACM Int. Conf. Software Engineering (ICSE)*, Lisbon, Portugal, 2024.
[19] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, 2014, pp. 437–440.
[20] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, and A. Blanco, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, Feb. 2021.
[21] Y. Liu *et al.*, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2023.