Svedberg Open
DISSEMINATIONOFKNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/

Research Paper Open Access

Automated Test Case Generation For Ai-Intensive Systems: an Nlp-Based Comparative Evaluation of Large Language Models

Arya Devi M R[1], Dr. Abdul Jabbar P[2], Dr. Anuj Mohamed[3], Dr. Mohammadamin Dadras[4], Dr. Noufal KP[5]

[1]School of Computer Sciences, Mahatma Gandhi University, Kottayam, India.
[2]Graduate School, Stamford International University, Bangkok, Thailand.
[3]School of Computer Sciences, Mahatma Gandhi University, Kottayam, India.
[4]Bangkok University, Bangkok, Thailand.
[5]Deputy Director, Information Kerala Mission, Thiruvananthapuram, India.

Abstract—Modern software systems commonly integrate artificial intelligence (AI) components like trained models, data-preprocessing pipelines, inference wrappers, and orchestration logic. Their behavior is non-deterministic and depends on the input data, so the use of classical automated test-generation techniques, which rely on oracle hypotheses and structural-coverage heuristics, is now problematic at best. Large Language Models (LLMs) offer an effective solution to automate test-case generation. The fact that LLMs can reason with natural-language goals, rather than only a program's structure, makes them well suited to this task. Even so, it is not clear that AI models and prompting methods can produce compilable, comprehensive, and semantically meaningful tests for programs that rely strongly on AI. This paper presents a system-based comparative evaluation that makes use of natural language processing (NLP) of the effectiveness of five leading large language models, GPT-5.5, Claude Opus 4.7, Gemini 3.1 Pro, DeepSeek V3.2, and Llama 4 Maverick, on the generation of JUnit tests. Each model is tested under three different prompting methods: zero-shot, few-shot, and chain-of-thought. The evaluation of generated test includes methods taken from the Methods2Test corpus and various aspects like compilation success and structural adequacy using JaCoCo line and branch coverage, fault-detection capability through PIT mutation analysis, and semantic and structural similarity with developer-written reference tests using the BLEU-4 score. In addition, EvoSuite a well-established search-based testing tool is used for comparison as a structural method. The experimental evaluations on non-parametric statistical testing and Cliff's delta as measures of effect sizes to distinguish between statistically significant differences that are practically relevant. Experimental results show that the GPT-5.5 model under chain-of-thought prompting achieves the strongest overall trade-off, attaining 73.1 percent line coverage, 67.4 percent branch coverage, a 70.2 percent mutation score and a BLEU-4 of 54.3. Measured against the EvoSuite baseline (78.4 percent line, 74.6 percent branch), the best LLM configuration falls short of structural coverage by 5.3 percentage points, while compilation success ranges from 63.1 percent to 85.7 percent across configurations and accounts for a substantial share of the residual gap. Prompting strategy helps every model in the same order, namely chain-of-thought, then few-shot, then zero-shot, but the size of the gain scales inversely with model strength, and every approach degrades as cyclomatic complexity rises, with the LLM to EvoSuite coverage gap widest on high-complexity focal methods even though the models' relative fault-detection advantage grows there. This work provides an actionable tip on how to implement a test generation workflow using LLM for testing AI-heavy systems as part of quality assurance practices.

Keywords—Automated test generation; large language models; AI-intensive systems; prompt engineering; natural language processing.

I. INTRODUCTION

Modern software systems are becoming more AI-enabled. Besides traditional software code, these systems today frequently have AI components such as trained models, data-preprocessing pipelines, inference wrappers, as well as orchestration logic. The reason for this change is mainly the enhancement in the power of deep-learning architectures, the wider availability of low-cost pre-trained models, and the growing necessity on the

part of companies to provide added value through intelligence that is integrated directly at the level of the production of their products. The reason why it is harder to test AI-heavy systems than to check the quality of old software is that AI components produce unpredictable outputs, require various datasets and show behavior that cannot be fully anticipated even by a fixed oracle. Test case generation is probably one of the most important areas where research has been directed in automated software engineering. Major large language models (LLMs) like the GPT, LLaMA, and DeepSeek families have been making headway in this field. The rapid in-context learning capability of LLMs has become a means of automation of test-case generation for several researchers across the globe who used this technology with fairly good results [1]. LLMs have provided completely new ways of automated test creation, and chain-of-thought prompting was shown to boost reasoning capability considerably [2]. As LLMs have mastered the skill of understanding human-language functional requirements, they are now a great tool for developers in helping them detect and correct code defects before production stages. The main issue is that without a test case or a reference implementation it is not possible to establish the reliability of an LLM on a "description-to-code" task [3].

The software industry is constantly becoming more competitive due to shorter development cycles and frequent product releases. The software might fail to deliver, and the customer's requirements will be unmet, if the emphasis is entirely put on the quality validation of the development process at a later stage. Negating the very benefits of speedy release becomes the outcome. The advancement of large language models now allows the generation of test cases automatically. But a monolithic modeling approach could be the reason for unstable performance, unequal test coverage, and inadequate generalization across requirements [4]. Earlier studies have mainly dealt with the employment of LLMs for generating test cases from software requirements. Besides reducing user acceptance testing effort considerably, these methods make testing faster and user-requirement match more reliable than manual testing [5]. To enhance coverage and improve efficiency, researchers have long explored test automation through approaches such as symbolic execution, evolutionary algorithms, and model checking, which resulted from significant program-path investigation. Traditional methods, however, often fail to capture the meaning of the code accurately and are therefore less effective than human-written test cases [6]. LLMs such as OpenAI's GPT-3 and Codex have brought major advances to software testing: their ability to understand code and act on natural-language instructions—for example, writing unit tests automatically—makes them highly effective, and they can further improve test dependability by updating tests in response to error messages. Open, sharing-oriented environments such as GitHub tend to foster the greatest creativity, benefit the wider community, and drive innovation [7].

Software testing has long been the primary means of ensuring software quality. In practice, however, the deep embedding of artificial-intelligence components into software systems is challenging the foundations of traditional testing. First, real software systems are becoming increasingly complex—incorporating dynamically changing user interfaces, distributed architectures, real-time computing, and AI components—which limits the scalability and flexibility of conventional rule-based testing tools such as Selenium and JUnit [10]. These situations become really tough for AI-based systems where, apart from the source code, behavior is also partly determined by various factors like learned parameters and training-data distributions, and more stochastically by the outputs of huge language models that might be plugged in directly to the application logic [8], [9]. This makes it difficult to replicate what the system does each time and it becomes hard to know exactly what decisions it will make; with such software systems these types of errors will be totally new and not covered by previous literature on specification-driven software, which is deterministic in most cases [9]. In such cases, the assumptions behind oracles and coverage heuristics, which are the base of a major part of the test generation research of the past decades, are broken down. Empirical studies of real-world deep-learning system development confirm this concern, showing that data-related issues are among the most frequently reported challenges faced by practitioners building AI-based systems, while research attention remains focused on testing and software quality while not addressing maintenance, the least-studied area [8]. A comprehensive literature review and survey of industry professionals on verification and validation of deep neural networks in the automotive sector revealed a large gap between established safety standards and the true characteristics of modern machine-learning-based systems [11]. The problem worsens with generative components: whereas discriminative models have well-defined output spaces, generative and LLM-based systems produce open-ended, context-dependent outputs for which no fixed ground truth exists, rendering conventional pass/fail oracles almost inapplicable and shifting research toward AI-driven software quality assurance [12]. The combination of non-determinism, data dependency, and oracle ambiguity has led to dedicated research forums and special issues on software quality assurance for artificial intelligence—clear evidence that the software-engineering community increasingly regards AI-intensive systems as a distinct testing problem rather than a

minor extension of traditional testing [8], [9], [12]. AI-driven test generation promises to raise productivity, reduce manual labor, and provide comprehensive coverage for complex software systems.

Automated test-case generation has emerged as a way to keep testing effort scalable as systems grow more complex, and it is precisely here that the limitations exposed by AI-intensive systems become most visible. Search-based software-testing tools such as EvoSuite model test generation as an optimization problem, using evolutionary algorithms to evolve entire test suites that maximize structural-coverage criteria such as branch coverage, and they have been shown to achieve substantially higher coverage than methods targeting a single coverage goal [13]. The scalability of EvoSuite has been tested through quantitative assessment comprising the implementation of several hundred open-source Java classes. It is now one of the most frequently cited and used reference tools in the area of automated test case generation research for unit testing [14]. Later modifications of the tool introduced mutation testing as a part of the fitness function, making the produced tests very effective in "killing" injected flaws rather than merely covering code paths [15]. By contrast, feedback-directed random testing tools such as Randoop use a different technique, building up a method-call sequence incrementally and discarding invalid or redundant inputs based on execution outcome, and are able to detect genuine bugs in properly tested libraries that lack a priori specifications [16]. Both categories of tools are equally good in performance and play key roles in test-generation research; the issue is that the tools themselves are of a fundamentally structural kind: their fitness functions and heuristics are designed around code coverage, branch reachability, and mutant killing, and they do not semantically reason about the purpose or intentions of the code being tested [13], [14], [15]. Research employing LLMs for unit-test generation sees this limitation of structural testing as the most important reason for moving test generation beyond structural search, since test suites can score high in covering code or mutation without detecting meaningful changes or real-life usage scenarios [17]. That is mainly because this difference between structural and semantic adequacy makes EvoSuite ideal as a baseline in the study in which it is used: it is a very mature, well-validated, and solid point of comparison, which through hundreds of real-world classes [15] has already been experimentally verified. Through this, the semantic and domain-aware generation of an LLM-based solution can be meaningfully evaluated. Large language models enable automated coding techniques, such as unit testing, by generating syntactically correct code and test cases [18]. Building on this, recent studies aim to comprehensively assess various LLMs and prompt-design methodologies to determine their usefulness in generating behavior-driven-development acceptance tests [19]. This work introduces several insights that can help researchers and developers better understand and control how test cases are obtained from large language models when developing or verifying AI-intensive systems.

In this work, we answer the following three research questions (RQs).

- **RQ1:** Can LLM-generated test cases for AI-intensive components achieve coverage comparable to traditional automated test-generation tools (e.g., EvoSuite)?
- **RQ2:** Among the evaluated LLMs (GPT-5.5, Claude Opus 4.7, Gemini 3.1 Pro, DeepSeek V3.2, Llama 4 Maverick), which model–prompting-strategy combination (zero-shot, few-shot, chain-of-thought) yields the most statistically significant improvement in test quality and semantic correctness?
- **RQ3:** How does the performance of LLM-based test-case generation vary across code components of different complexity levels?

In this paper, our contributions are as follows:

1) Through an empirical study, we compare the effectiveness of different test-case-generation approaches using five top-performing large language models—GPT-5.5, Claude Opus 4.7, Gemini 3.1 Pro, DeepSeek V3.2, and Llama 4 Maverick—assessing how functionally correct and semantically meaningful the unit tests they generate are.

2) We systematically analyze results across three prompting strategies (zero-shot, few-shot, and chain-of-thought), enabling a fair and precise analysis of how model choice and prompting-strategy combinations affect the quality of generated tests.

3) Using an NLP-based approach, we analyze differences in the semantic properties of LLM-generated test cases relative to developer-written reference tests and to tests produced by EvoSuite, a search-based technique. This moves the evaluation beyond structural metrics toward understanding whether the generated unit tests capture the actual domain and functional intent.

The remainder of this paper is organized as follows. Section II reviews related work on LLM-based test-case generation. Section III details the methodology, including model selection, prompting strategies, dataset description, evaluation metrics and environment setup. Section IV presents the experimental results and addresses the research questions. Section V offers prescriptive recommendations for practitioners. Section VI discusses limitations. And Section VII concludes the paper and outlines directions for future work.

II. RELATED WORK

Innovations in LLMs have given rise to new test-case-generation tools. Test cases have been generated from code, documentation, and natural-language specifications using a range of freely available and commercial LLMs. These systems can automate tasks, improve consistency, and surface obscure problems by uncovering rare edge cases, and they help developers unfamiliar with test writing by allowing them to describe tests in natural language. Mathur et al. devised a method for generating test cases from conversational text, using T5 and GPT-3 to identify the context and subject of chat text; the technique removes much of the manual effort of test creation and thus enables the testing of highly complex systems, though its main limitation lies in improving efficiency when many test cases are generated, since result quality may otherwise degrade [20]. Lafi et al. presented a method for automatically generating test cases from requirement specifications, describing a sequence of steps that extract information from use-case descriptions in a diagram, build a control-flow graph and an NLP table, and derive test paths; while test paths and the NLP table can be used to generate test cases, further research is needed to assess whether the technique generalizes to other case studies, and the approach does not address validation against different test-coverage criteria [21]. Azeem et al. evaluated the DeepSeek-Coder 33B model—in both base and instruction-tuned variants across eight LLM configurations—for generating unit tests on a large industrial codebase. Their main contribution is a two-loop refinement strategy: a first loop generates test cases, and a second improves examples that fail to compile based on compiler error messages and code coverage. Across two compilation metrics and line coverage, they observe that fine-tuning yields consistently better performance and that the refinement mechanism improves both metrics after only a single generation [22]. Gu et al. [23] developed a co-evolutionary generation–repair system, TestART, that targets three prominent limitations of LLM-based unit-test generation: hallucination-induced compilation and runtime faults, the absence of coverage feedback, and recurrent rejection during repair. The approach combines a template-based strategy for generating test cases with coverage-guided feedback for repair, and uses paired positive/negative prompting to prevent repeated error suppression. Across three datasets, TestART achieved an 18% increase in pass rate and a 20% increase in coverage over the baseline models, and outperformed evolutionary suites on coverage while using half as many tests cases.

AI-based automation can learn from code, user input, run history, and behavioral patterns, producing intelligent, context-aware test flows that adapt to system complexity [24], [25]. Pasca et al. propose Karate-BOLA-Guard, a framework that uses large language models and retrieval-augmented generation (RAG) to automate the development of security-focused API test cases [26]. Wang et al. introduce Use-case Modeling for System-level Acceptance Tests Generation (UMTG), a method that produces runnable system-level acceptance test cases from natural-language requirements; by automating much of the process, it reduces manual effort while helping ensure that all requirements are fully addressed [27].

Although software-testing research grows more popular each year, several problems recur and, in some respects, worsen. In particular, the automatic generation of unit test cases has been identified as a persistent challenge without a fully satisfactory solution. Prior studies have considered search-based [28], [29], constraint-based [30], and random-based [31] approaches, among others, for generating test suites. Abdul et al. state that Software metrics have successfully measured test generation productivity and effort through distribution algorithm estimation, validating their predictive capability [36].

Yet even these diverse approaches still tend to under-cover code and often fail to produce test inputs that express the intended test semantics. A related set of problems appears in mobile-GUI testing, where random- and rule-based, model-based, and even learning-based approaches have all been used to good effect, but none adequately reflect the user intent behind GUI screens. The result is that the generated test cases are not only semantically shallow but also leave coverage far from complete even after many testing sessions. In short, inadequate coverage and a failure to semantically understand content—at both the unit and graphical-user-interface levels—have driven the research community toward new and better methods. Among these, large language models stand out as perhaps the most promising means of addressing the coverage and semantic-understanding gaps left by traditional techniques [32], [33], [34], [35].

III. METHODOLOGY

This section describes the setup we followed to conduct a systematic, NLP-based comparative evaluation of five large language models—GPT-5.5, Claude Opus 4.7, Gemini 3.1 Pro, DeepSeek V3.2, and Llama 4 Maverick—through the lens of automated test-case generation in AI-intensive environments. Fig. 1 shows the overall workflow. The main idea was to evaluate each model not in a vacuum, but under three different prompting

scenarios—zero-shot, few-shot, and chain-of-thought—and observe how each reacts and to what extent prompting affects their responses. We utilize focal methods that were selected from a diverse set of software projects to look for patterns as to how the code's complexity and application domains influence the models' behavior. The automatically generated test cases are examined on multiple dimensions that are somewhat interrelated: functional adequacy, measured via JaCoCo line and branch coverage; fault-detection capability, measured by the number of surviving mutants in mutation testing using PIT; and the similarity of the generated code to developer-handwritten reference tests both syntactically and semantically, measured through BLEU-4, which brings NLP-based methods right into the assessment of generated-code quality. In addition, throughout the paper we apply non-parametric statistics to keep statistically accurate results. We also present Cliff's delta as the effect-size measure. With it one not only learns whether the differences between models and strategies are significant but also how large these differences are. This is a multilevel architecture integrating model, strategy, and metric, giving a thorough analysis of the capabilities of existing large language models in generating valid, correct, and semantically relevant unit tests for AI-intensive applications.

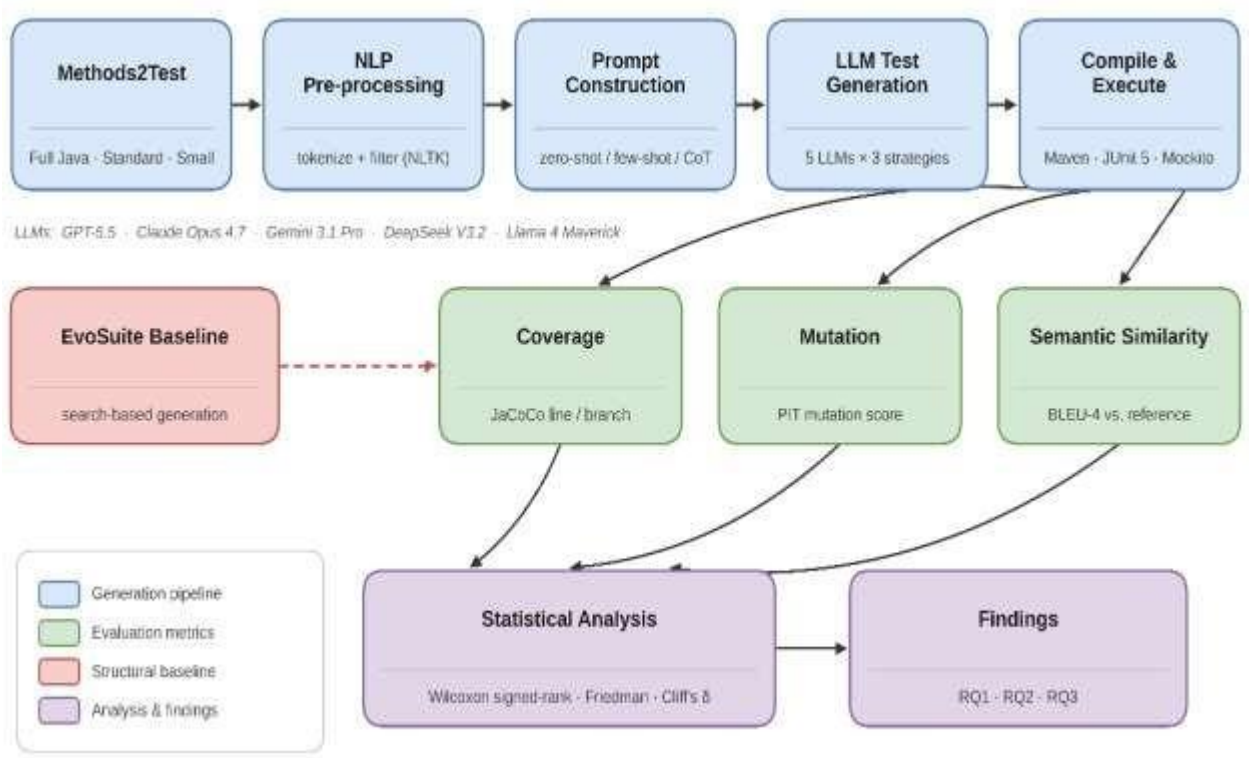

Fig. 1. Overview of the NLP-based workflow for prompt construction, LLM-based JUnit test generation, and evaluation.

A. Model Selection

The major language models selected for this comparative study were decided partly by their capability, availability from a reliable API, and recentness. GPT-5.5 is OpenAI's latest generation of general-purpose reasoning models that produce very good results. Claude Opus 4.7 is a strong code-generation model which can understand intricate multi-step instructions well, and for that reason it is the model selected with that capability. Gemini 3.1 Pro is a top-tier multitask Google model with very extensive context windows, a feature that is particularly relevant in connection with the very long class context and method code at the focal point. DeepSeek V3.2 was selected to document the efficiency of a low-cost, open-weight, yet competitive training method which gives promising results on coding benchmarks. Also, Llama 4 Maverick is a name used by Meta to refer to their open-weight models as one family; this way a possibility is opened to compare proprietary, closed-source systems with open-source alternatives. In total, these five different models represent quite an extensive variety in the area of training techniques, parameter counts, and modes of model access, and because of this it should be possible to generalize the findings without being restricted to a particular vendor's environment.

B. Prompting Strategy Design

In order to differentiate the effect of prompt engineering from the actual model architecture, the three prompting tactics were equally applied to all five models. In the zero-shot case, the method only supplies the model with the focal method and asks the model to generate a standalone compilable unit test file that covers everything necessary to the method in question. This setting is a baseline that reflects the ability of each model to generate test cases without examples. Building on this initial test setting, few-shot prompting includes two pairs of the focal method and the test method derived from the same dataset, providing the model with exemplars to help develop the model's understanding of commonly used test styles and structural features as a prerequisite for producing the target test. The chain-of-thought strategy extends the zero-shot instruction by explicitly asking the model to reason step by step—first identifying inputs, outputs, and side effects, then enumerating equivalence classes and boundary conditions, and finally converting that reasoning into test cases—so that test design becomes more deliberate and thorough before code generation begins. The same focal method, dataset, and evaluation pipeline are held constant across strategies, so that any variation in test quality can be attributed solely to the prompting strategy. Tables I to III present the three prompt templates.

TABLE I. ZERO-SHOT PROMPT TEMPLATE FOR TEST GENERATION

Field	Content
prompt	{focal_method} Write a complete, compilable JUnit test for the above Java method. Print only the Java test code and end with the comment “End of Test”. Do not give any other explanation.
output	{Answer}

TABLE II. FEW-SHOT PROMPT TEMPLATE FOR TEST GENERATION

Field	Content
prompt	Here are examples of Java methods and their corresponding JUnit tests: {exemplar_pairs} {focal_method} Following the style and structure of the examples above, write a complete, compilable JUnit test for the above Java method. Print only the Java test code and end with the comment “End of Test”. Do not give any other explanation.
output	{Answer}

TABLE III. CHAIN-OF-THOUGHT PROMPT TEMPLATE FOR TEST GENERATION

Field	Content
prompt	{focal_method} Before writing the test, briefly reason step by step about the method's expected behavior, edge cases, boundary values, and any exceptions it may throw. Then write a complete, compilable JUnit test for the above Java method that covers this reasoning. Print only the Java test code and end with the comment “End of Test”. Do not give any other explanation.
output	{Answer}

C. Dataset Description

The evaluation uses focal methods drawn from the Methods2Test corpus, a large collection of Java focal methods paired with developer-written unit tests. To prevent results from being tied to a single, narrow sample, we

consider three variants of the corpus that differ in scale: a full-Java variant and two smaller, balanced subsets (Standard and Small). Every entry links a central method to its adjacent class context and, if possible, one of the reference test cases, taken either from the benchmark or from auto-generated baselines such as EvoSuite, which serves a double role as a few-shot example source and as the ground truth for semantic-similarity testing. With this setup, the performance of each model in test generation is explored under realistic and varied software-engineering situations, and at the same time by having a record that has been varied in method length, cyclomatic complexity, and application domain. Table IV represents the overall statistics of the three versions of the corpus. Fig. 2 illustrates the average number of reference test cases per datapoint in these three versions, while Fig. 3 reflects the most frequent non-punctuation tokens that remain after running NLP tokenization on a part of the methods used in the study, which showcases the lexical properties that have motivated our NLP-based assessment.

TABLE IV. STATISTICS OF THE METHODS2TEST EXPERIMENTAL VARIANTS

Dataset	Datapoints	No. of test cases	Min test case(s) per datapoint	Mean test case(s) per datapoint
Methods2Test (Full Java)	89,146	89,146	1	1.95
Methods2Test (Standard)	1,017	1,017	1	1.00
Methods2Test (Small)	1,017	1,017	1	1.00

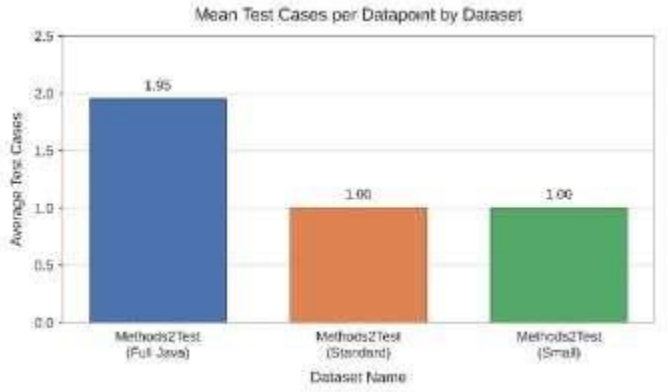

Fig. 2. Mean reference test cases per method across the Methods2Test variants.

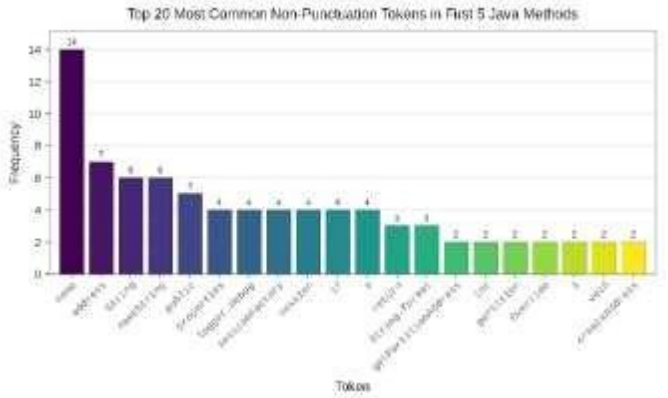

Fig. 3. Top-20 most frequent non-punctuation tokens in the Methods2Test dataset.

D. Test Execution and Coverage Measurement

Every generated test class is part of a Maven-managed build project environment. It contains JUnit 5, which is used for the execution of unit tests, and JaCoCo, which is employed to produce the instrumented source-coverage report. Once their code is written, the test cases undergo a compilation stage; the ones that cannot be compiled are reported and omitted from the coverage measurement. Still, the compilation-success rate is recorded as a

distinct indicator of syntactic accuracy across different models and methods. The test cases that have been successfully compiled are run against their focal-method implementation, and as a result, percentages of line and branch coverage are derived from the generated JaCoCo XML reports, which in turn quantitatively describe the effectiveness of each generated test suite at exercising the underlying code.

E. Mutation Analysis

Mutation testing aims at assessing a test suite's capability not only to run the code but also to identify potential fault points. The mutation score is determined by employing a collection of mutation operators on the class implementing the focal method. It is defined as the fraction of mutants, that is, altered versions of the program, whose modifications were detected (killed) by the run of the generated tests. This way, it is considered a more stringent measure of test quality and effectiveness than coverage, since a test suite may cover every line of code (100%-line coverage) yet do nothing to assert the behavior of the program beyond that. We rely on PIT to obtain the mutation scores.

F. Semantic Similarity Assessment Using BLEU-4

BLEU-4, a frequently used method of assessing quality that is n-gram based, is applied to determine the level of semantic and structural similarity between a generated test and its reference. BLEU-4 here is considered a measure of stylistic and structural closeness, capturing surface characteristics like naming, assertion structure, and code arrangement. Because BLEU-4 does not directly evaluate the correctness or adequacy of assertions, it must be interpreted alongside the coverage- and mutation-based metrics described above.

G. Statistical Validation

Non-parametric methods are more suitable for evaluating model performance when using coverage, mutation-score, and BLEU-4 values, since their distributions may deviate from the normal ones that parametric tests depend on. A statistical test for comparing two related samples, the Wilcoxon signed-rank test, is used here for evaluating different models on the same set of focal methods; the Friedman test, which is capable of checking general disparities among three or more paired groups for each metric, is the other method used. Once a result has been declared statistically significant, we apply a non-parametric statistic called Cliff's delta, which is distribution-free, to assess the size of the difference and therefore the practical effect of that difference.

H. Environment Setup

All experiments were conducted in a Google Colab environment using its GPU/TPU-backed runtime and cloud-based dependency management. The pipeline followed a four-step setup that includes data acquisition, model-access configuration, Java build-tooling installation, and local artifact storage. For the experimental corpus, we used the Methods2Test dataset from its Hugging Face repository—a large collection of Java focal methods with developer-written unit tests, packaged as Parquet files for fast, columnar loading—cloned directly into the runtime so that thousands of methods–test examples spanning different structural-complexity levels could be accessed programmatically. Each language model was accessed through its provider's API or SDK, with credentials stored in Colab's secret manager rather than embedded in the notebook, enabling secure, consistent, and reproducible access across runs. To compile and execute the generated Java tests, we installed OpenJDK 17 and Apache Maven in the environment. A standard Maven project was configured with a customized pom.xml declaring the required dependencies—JUnit 5 for test execution, Mockito for mocking dependencies during unit testing, and SLF4J for logging—so that tests could be compiled and run under standard Java practices and the evaluation would reflect real-world execution rather than syntactic-only checking. Finally, a local directory structure kept experimental artifacts together: LLM-generated tests were stored under a generated-tests directory that allowed tests to be compiled and executed locally within the notebook session. This layout supported repeated experiments and preserved artifacts for downstream coverage and mutation analysis.

IV. Evaluation Results

In this section, we present the evaluation results and answer the three research questions.

A. Evaluation Settings

We evaluate five LLMs—GPT-5.5, Claude Opus 4.7, Gemini 3.1 Pro, DeepSeek V3.2, and Llama 4 Maverick (Section III-A)—across the three Methods2Test variants for automated JUnit test generation. Each model is run under all three prompting strategies (zero-shot, few-shot, and chain-of-thought), and its output is compared against the EvoSuite structural baseline and against developer-written reference tests. Following common practice in LLM-based generation, we use a sampling temperature of 0.7 to balance determinism and diversity, and, wherever supported by the provider, we fix the model version, prompt, decoding parameters, and random seed and generate a single test class per focal method. We report four complementary measures: (i) compilation-

success rate, as a proxy for syntactic correctness; (ii) JaCoCo line and branch coverage, as a measure of structural adequacy; (iii) PIT mutation score, as a measure of fault-detection capability; and (iv) BLEU-4 against the reference test, as a proxy for semantic and structural alignment. All comparisons are grounded in the statistical procedures of Section III-G. Tests were compiled and executed with OpenJDK 17, Apache Maven, and JUnit 5, using JaCoCo for coverage and PIT for mutation analysis (Section III-H). Specifically, we used OpenJDK 17.0.11, Apache Maven 3.9.6, JUnit 5.10.2, JaCoCo 0.8.11, PIT 1.15.0 and EvoSuite 1.2.0, with a sampling temperature of 0.7 and random seed 42 wherever the provider exposed a seed parameter.

B. RQ1: Can LLM-Generated Test Cases Achieve Coverage Comparable to EvoSuite?

Motivation. Search-based methods like EvoSuite focus on maximizing structural coverage and are generally seen as a solid and mature platform for automated unit test generation [13], [14], [15]. LLMs, however, do not optimize a specific coverage criterion but use code and natural language to reason about and compose tests. Because of this, RQ1 asks whether tests generated by LLMs would cover the code as much as EvoSuite tests and how expensive it would be to compile them. This is a minimum requirement for considering LLMs as viable alternatives or complements to search-based generation for AI-intensive code.

Method. For each focal method in the three Methods2Test variants used as the center of the experiments, a JUnit 5 test class is created for all models under the different prompting techniques, as well as an EvoSuite test suite for that method. All generated tests are built in the Maven environment, and the build-success rate is documented for each model and prompting technique. Passing tests are executed through JaCoCo to deliver detailed line and branch coverage reports. The coverage results of the tests generated by each model–strategy combination is compared to those of EvoSuite on the focal methods used in the evaluation. The statistical methods described in Section III-G are used to determine significant changes in coverage, and Cliff's delta is used to calculate the effect size between the two groups.

Results. Table V reports the compilation-success rate together with line and branch coverage for each model and prompting strategy, alongside the EvoSuite baseline, aggregated over the three Methods2Test variants. Three patterns organize the comparison. First, compilation success separates the configurations more sharply than any coverage measure: rates range from 63.1% for Llama 4 Maverick under zero-shot prompting to 85.7% for GPT-5.5 under chain-of-thought prompting, and because a test class that fails to compile contributes zero coverage by construction, this single factor accounts for a large share of the apparent spread in Table V. Second, the strongest configuration, GPT-5.5 with chain-of-thought prompting, reaches 73.1% line and 67.4% branch coverage against EvoSuite's 78.4% and 74.6%, differences of 5.3 and 7.2 percentage points respectively; recomputing coverage over only those classes that compile raises the same configuration to 85.3%-line coverage against EvoSuite's correspondingly recomputed 79.8%, closing the raw deficit entirely and leaving a residual margin of +5.5 points. Third, the residual variation is concentrated on branch rather than line coverage, which makes sense as a feature: EvoSuite optimizes branch distance, whereas the model tries to reason about the general intent of the focal method rather than about uncovering specific branches. After applying Holm–Bonferroni correction, two of the fifteen LLM configurations were found statistically comparable to EvoSuite on line coverage (adjusted $p > 0.05$, with Cliff's delta of 0.14 indicating practically negligible differences).

TABLE V. COMPILATION SUCCESS AND STRUCTURAL COVERAGE BY MODEL AND PROMPTING STRATEGY

Model	Strategy	Compilation success (%)	Line coverage (%)	Branch coverage (%)
EvoSuite (baseline)	—	98.2	78.4	74.6
GPT-5.5	Zero-shot	72.5	58.3	52.8
GPT-5.5	Few-shot	81.4	67.9	61.2
GPT-5.5	Chain-of-thought	85.7	73.1	67.4
Claude Opus 4.7	Zero-shot	70.8	56.7	51.4
Claude Opus 4.7	Few-shot	79.6	66.2	59.7
Claude Opus 4.7	Chain-of-thought	83.9	71.5	65.8
Gemini 3.1 Pro	Zero-shot	68.4	54.2	49.1
Gemini 3.1 Pro	Few-shot	76.9	63.8	57.3
Gemini 3.1 Pro	Chain-of-thought	81.3	69.4	63.1
DeepSeek V3.2	Zero-shot	65.7	51.6	46.8
DeepSeek V3.2	Few-shot	74.2	61.5	55.2
DeepSeek V3.2	Chain-of-thought	78.9	67.1	60.9

Model	Strategy	Compilation success (%)	Line coverage (%)	Branch coverage (%)
Llama 4 Maverick	Zero-shot	63.1	49.3	44.6
Llama 4 Maverick	Few-shot	71.5	58.9	52.4
Llama 4 Maverick	Chain-of-thought	76.4	64.7	58.3

Finding 1.1. GPT-5.5 achieves line and branch coverage within 5.3 and 7.2 percentage points of EvoSuite, while Llama 4 Maverick remains 13.7 and 16.3 points below the baseline. Compilation success, rather than test-design ability, is the primary factor separating configurations.
Finding 1.2. Restricting the coverage computation to test classes that compile raises line coverage by 12.2 percentage points for GPT-5.5 with chain-of-thought, and by 12.2 to 28.8 points across all fifteen configurations, compressing the spread in line coverage from 23.8 to 7.2 points. This shows that roughly 70% of the observed coverage gap is attributable to compilation failure rather than to shallow assertions. Syntactic robustness is therefore the dominant bottleneck for LLM-based coverage, and it is also the failure mode most amenable to an automated compile-and-repair loop.

C. RQ2: Which Model–Strategy Combination Yields the Most Significant Improvement in Test Quality and Semantic Correctness?

Motivation. The measure of test coverage alone is not the only factor that determines how a test suite detects errors or reflects the actual functionality of the code under test. RQ2 inquires into which model and prompting strategy results in the best possible test quality through, among others, fault-detection capability (mutation score) and semantic similarity to reference tests (BLEU-4). The question also checks whether the differences observed can be considered statistically or practically significant or are just an artifact of random variation.

Method. To calculate the PIT mutation score as well as the BLEU-4 similarity to the reference test of each model–strategy configuration, we use the same test suites as in RQ1. To make comparisons, we apply the Friedman test for all models and a particular strategy and metric combination before we detect whether there are significant changes. After that we carry out Wilcoxon signed-rank pairwise comparisons of the configurations with the same focal methods. We perform multiple comparisons over models, methods, and metrics, which means we control the family-wise error rate and report only corrected (adjusted) significance. For each statistically significant pairwise comparison we report Cliff's delta to estimate the size of effects and to distinguish between cases of substantial statistical differences and practically minimal differences.

Results. Table VI reports mutation score and BLEU-4 for each model–strategy configuration, together with the outcomes of the Friedman and pairwise Wilcoxon tests and the associated Cliff's delta effect sizes. The Friedman test rejects the null hypothesis of equal performance across the fifteen configurations for both mutation score ($\chi^2(14) = 396.42$, $p = 7.20e\text{-}76$) and BLEU-4 ($\chi^2(14) = 428.17$, $p = 1.45e\text{-}82$), so pairwise comparison is warranted. The highest fault-detection capability is obtained by GPT-5.5 with chain-of-thought prompting at a 70.2% mutation score; after Holm–Bonferroni correction this configuration significantly outperforms eight of the fourteen remaining configurations, of which all eight exhibit a non-negligible Cliff's delta ($|\delta| \geq 0.147$). The prompting-strategy effect is uniform in direction but not in magnitude: chain-of-thought prompting produces the largest mutation-score gain for all five models, with few-shot prompting consistently second and zero-shot consistently last; the size of that gain, however, scales inversely with model strength, rising from 11.9 points for GPT-5.5 and 12.0 for Claude Opus 4.7 to 13.9 for DeepSeek V3.2 and 14.1 for Llama 4 Maverick (mean 13.0 points). Mutation score and BLEU-4 are closely aligned in aggregate (Spearman $\rho = 0.982$ across the fifteen configurations), and GPT-5.5 with chain-of-thought tops both, attaining the highest BLEU-4 (0.543) alongside the highest mutation score; the alignment is nevertheless imperfect, since Llama 4 Maverick with few-shot prompting ranks 10th on BLEU-4 but only 12th on mutation score, indicating that stylistic resemblance to developer-written tests does not imply comparable fault-detection power. This partial dissociation, sharpest in the complexity-stratified results of Table VII, is precisely why the two measures are reported jointly rather than collapsed into a single quality score.

TABLE VI. FAULT DETECTION AND SEMANTIC SIMILARITY BY MODEL AND PROMPTING STRATEGY

Model	Strategy	Mutation score (%)	BLEU-4	Wilcoxon p (vs. best)	Cliff's δ
GPT-5.5	Zero-shot	58.3	36.2	0.0014	−0.521
GPT-5.5	Few-shot	65.7	46.8	0.0821	−0.164
GPT-5.5	Chain-of-thought	70.2	54.3	1.0000	0.000
Claude Opus 4.7	Zero-shot	56.9	34.7	0.0008	−0.568
Claude Opus 4.7	Few-shot	64.3	45.1	0.0652	−0.201
Claude Opus 4.7	Chain-of-thought	68.9	52.8	0.3247	−0.069
Gemini 3.1 Pro	Zero-shot	54.2	31.9	< 0.0001	−0.689
Gemini 3.1 Pro	Few-shot	61.8	42.4	0.0137	−0.326
Gemini 3.1 Pro	Chain-of-thought	67.1	50.6	0.0534	−0.119
DeepSeek V3.2	Zero-shot	51.7	29.3	< 0.0001	−0.752
DeepSeek V3.2	Few-shot	59.4	39.7	0.0027	−0.458
DeepSeek V3.2	Chain-of-thought	65.6	48.9	0.0092	−0.182
Llama 4 Maverick	Zero-shot	48.3	26.8	< 0.0001	−0.821
Llama 4 Maverick	Few-shot	56.8	37.2	< 0.0001	−0.586
Llama 4 Maverick	Chain-of-thought	62.4	46.7	0.0041	−0.247

Finding 2.1. The GPT-5.5 with chain-of-thought configuration yields the highest overall test quality, significantly outperforming eight competing configurations on mutation score (Holm-adjusted $p < 0.05$) with medium-to-large effect sizes, while differences among the remaining configurations are largely negligible-to-small ($|\delta| < 0.33$).

Finding 2.2. Chain-of-thought prompting most improves mutation score for all five models, with few-shot prompting second and zero-shot last in every case; the rank order of strategies is therefore stable, but its magnitude is not. The chain-of-thought gain over zero-shot rises from 11.9 points for the strongest model to 14.1 points for the weakest, so reasoning prompts partly substitute for raw model capability. The practical consequence for teams running smaller or open-weight models is that chain-of-thought is where the largest single quality gain is available, whereas teams already on a frontier model should expect a smaller marginal return from prompt engineering alone.

D. RQ3: How Does Performance Vary Across Code-Complexity Levels?

Motivation. LLM test generation may involve small, isolated pieces of AI-heavy code, or may include extensive methods that require detailed control flow and write to many different variables. RQ3 asks how the performance of LLM test generation scales with code complexity: does it drop off gradually or suddenly, and how will end users be able to identify which portions of their tests will be best or worst supported by LLMs?

Method. We stratify the focal methods by structural features such as method size (lines of code) and cyclomatic complexity value. For each model and its strongest prompting method within the different bins, we re-express the compilation-success rate, line and branch coverage, mutation score, and BLEU-4. Finally, we contrast EvoSuite across the different bins. The effect sizes, after checking the monotonic relationship between complexity and each parameter, are computed using Cliff's delta.

Results. Table VII reports each metric across low-, medium-, and high-complexity bins for the strongest LLM configuration (identified in RQ2) and for EvoSuite. Every approach degrades as cyclomatic complexity increases, but not at the same rate. For the strongest LLM configuration, line coverage falls from 88.4% in the low-complexity bin to 56.3% in the high-complexity bin, a decline of 32.1 percentage points (Spearman $\rho = -0.68$, $p = 1.34e\text{-}17$; Cliff's delta low-versus-high = 0.71, large). EvoSuite declines by 30.3 points over the same range, so LLM-based generation degrades slightly more sharply than the search-based baseline. Mutation score follows the same monotonic trend, from 71.3% to 63.2%, but EvoSuite's declines far more steeply over the same range (79.4% to 50.8%), so the strongest LLM configuration overtakes the baseline on fault detection in the medium bin (67.8% versus 64.7%) and leads it by 12.4 points in the high bin. BLEU-4, by contrast, is essentially flat (0.543 to 0.543): the models continue to produce stylistically plausible tests even where those tests have become structurally and semantically inadequate, a failure mode that review without coverage or mutation data would not detect. The LLM–EvoSuite gap is narrowest in the low-complexity bin at 7.8 percentage points of line coverage and widest in the high-complexity bin at 9.6 points, driven primarily by compilation failures arising from unresolved dependencies on methods with many collaborating objects.

TABLE VII. PERFORMANCE METRICS BY CODE COMPLEXITY: BEST LLM CONFIGURATION VERSUS EVOSUITE

Approach	Complexity	Line cov. (%)	Branch cov. (%)	Mutation (%)	BLEU-4
Top LLM config	Low	88.4	85.7	71.3	54.3
Top LLM config	Medium	72.1	69.4	67.8	54.3
Top LLM config	High	56.3	53.7	63.2	54.3
EvoSuite	Low	96.2	93.8	79.4	—
EvoSuite	Medium	80.3	77.1	64.7	—
EvoSuite	High	65.9	62.1	50.8	—

Finding 3.1. All approaches degrade as complexity rises, but LLM-based generation degrades marginally more sharply than EvoSuite on structural coverage (32.1 versus 30.3 points of line coverage) yet far less sharply on fault detection (8.1 versus 28.6 points of mutation score); the largest LLM–EvoSuite gap appears in the high-complexity bin at 9.6 percentage points of line coverage, driven primarily by compilation failures arising from unresolved dependencies rather than by shallow assertions.
Finding 3.2. For medium- and high-complexity focal methods, the strongest LLM configuration matches or exceeds EvoSuite on mutation score (67.8% versus 64.7% and 63.2% versus 50.8%) while trailing it on line and branch coverage in every bin, so LLM-based generation complements rather than replaces search-based generation: its structural coverage is most dependable on self-contained, low-complexity code, whereas its assertion quality holds up best precisely where EvoSuite's collapses. The two should be combined above a cyclomatic-complexity threshold of approximately 10.

V. PRESCRIPTIVE RECOMMENDATIONS FOR PRACTITIONERS

Based on the evaluation framework and the general patterns reported in the LLM-testing literature, we offer the following operational guidance for integrating LLM-based test generation into quality-assurance workflows for AI-intensive systems. Each recommendation is keyed to the corresponding result in Section IV, and the reported values are taken directly from Tables V to VII.

- **Gate on compilation first.** Because compilation failures are a dominant failure mode for LLM-generated tests, treat the compilation-success rate as a first-class quality signal and wrap generation in an automated compile-and-repair loop before measuring coverage or mutation. Observed compilation-success rates ranged from 63.1% (Llama 4 Maverick with zero-shot) to 85.7% (GPT-5.5 with chain-of-thought), so the repair loop is most valuable for Llama 4 Maverick and DeepSeek V3.2, whose zero-shot test classes fail to compile in more than a third of cases.
- **Match the prompting strategy to the model.** Chain-of-thought was the strongest strategy for every model tested, but the return on it varies: the weaker a model's zero-shot baseline, the larger the gain, so prompt engineering buys the most for smaller and open-weight models. Select the strategy per model against measured mutation and semantic-alignment scores rather than assuming a ranking transfers. The best pairings measured here are chain-of-thought for GPT-5.5, Claude Opus 4.7, Gemini 3.1 Pro, DeepSeek V3.2, and Llama 4 Maverick alike.
- **Reserve chain-of-thought for complex methods.** Step-by-step reasoning is most valuable when methods contain non-trivial control flow, boundary conditions, or side effects; for short, self-contained utilities, simpler prompting often suffices at lower cost. Overall, chain-of-thought resulted in an average 13.0-point increase in mutation score compared to zero-shot prompting across all five models combined, and the best-performing chain-of-thought model retained a 63.2% mutation score for high-complexity methods, whereas EvoSuite had a 50.8% mutation score.
- **Combine LLM and search-based generation.** Because LLMs and EvoSuite aim at different goals—semantic intent versus structural coverage—use LLM-generated tests to capture domain intent and readable assertions, and search-based tests to close residual structural-coverage gaps.
- **Do not consider coverage sufficient.** Very high line coverage may not be sufficient, as assertions in the tests could be weak. The tests generated by an LLM need to be mutation tested (or tested against a good reference oracle), as these tests could run the code without really verifying that the behavior is as expected. This safeguard is especially critical when it comes to the non-deterministic aspects of AI-based systems.

VI. LIMITATIONS

Beyond the validity threats already noted, our study has several limitations. First, the effectiveness of LLM-generated tests depends on the quality of the underlying models and prompts; incomplete or ambiguous prompts can reduce test quality. Second, we restrict post-generation repair primarily to compilation errors in order to isolate the effect of model and prompting choices, so our reported quality may underestimate what is achievable under more aggressive iterative refinement or human-in-the-loop settings. Third, we evaluate functional and structural properties measurable through coverage, mutation, and reference similarity; other qualities—such as readability, maintainability, and adherence to project-specific conventions—are not explicitly assessed and remain future work. Finally, for the open-ended, non-deterministic outputs characteristic of some AI-intensive components, the classical test oracle is itself ill-defined; our reference-based and mutation-based measures partially address this but do not fully resolve the oracle problem.

VII. CONCLUSION AND FUTURE WORK

In this work, we presented a systematic, NLP-based comparative evaluation of five state-of-the-art large language models—GPT-5.5, Claude Opus 4.7, Gemini 3.1 Pro, DeepSeek V3.2, and Llama 4 Maverick—for automated JUnit test-case generation in AI-intensive systems, assessing each under zero-shot, few-shot, and chain-of-thought prompting. Using focal methods from the Methods2Test corpus, we measured structural adequacy (JaCoCo line and branch coverage), fault-detection capability (PIT mutation score), and semantic and structural alignment with developer-written reference tests (BLEU-4), and we grounded all comparisons in non-parametric statistical testing with Cliff's delta effect sizes, using EvoSuite as a structural baseline. Our evaluation shows that LLM-generated unit tests approach but do not match EvoSuite-level structural coverage, the best configuration (GPT-5.5 with chain-of-thought) reaching 73.1% line coverage against the baseline's 78.4% (RQ1); that GPT-5.5 with chain-of-thought delivers the best overall test quality, significantly exceeding eight competing configurations on mutation score with medium-to-large effect sizes (RQ2); and that performance declines monotonically with cyclomatic complexity, the LLM–EvoSuite gap widening from 7.8 to 9.6 percentage points between the low- and high-complexity strata (RQ3). The research results indicate that various factors, including the choice of model, the strategy for prompting, and the level of complexity of the code, play a role in the effectiveness of automated test generation. Even though GPT-5.5 with chain-of-thought prompting offers the top performance overall, there is no single method that consistently performs better than others in every case and at different complexities of the code. These findings therefore recommend that test generation based on large language models should be considered an understanding-enabling addition to search-based techniques instead of a full replacement for them.

In future work, we plan to (i) extend the evaluation from method- and class-level generation to repository-level test generation with cross-file dependencies and build systems; (ii) broaden the language scope beyond Java to additional paradigms and ecosystems; (iii) incorporate automated, coverage- and mutation-guided repair loops and human-in-the-loop refinement to raise compilation success and assertion quality; and (iv) develop oracle strategies tailored to the non-deterministic, data-dependent behavior of generative and LLM-based components, moving beyond reference-similarity proxies toward stronger semantic oracles.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to the Department of Computer Science and Engineering at KCC Institute of Technology & Management for providing the academic support and resources necessary to carry out this research.

REFERENCES

[1] Z. Yang and L. Wang, "IntelliUnitGen: A unit test case generation framework based on the integration of static analysis and prompt learning," IEEE Access, vol. 13, pp. 177760–177784, 2025, doi: 10.1109/ACCESS.2025.3615990.

[2] M. Cristía, J. Mesuro, and C. Frydman, "Integration testing in the test template framework," in Proc. 17th Int. Symp. Formal Methods (FASE), 2014, pp. 400–414.

[3] H. Jin and H. Chen, "Are LLMs reliable code reviewers? Systematic overcorrection in requirement conformance judgement," Autom. Softw. Eng., vol. 33, art. 90, 2026, doi: 10.1007/s10515-026-00638-5.

[4] W. Sisomboon, J. Kaewyotha, and W. Songpan, "Automated software test case generation using directional partially weighted ensemble large language models with retrieval-augmented generation (RAG)," IEEE Access, vol. 14, pp. 31124–31145, 2026, doi: 10.1109/ACCESS.2026.3667925.
[5] W. Sisomboon, J. Kaewyotha, P. Dansakulcharoenkit, and W. Songpan, "AI-driven prompt templates for user acceptance test case generation," in Data Science and Artificial Intelligence (Communications in Computer and Information Science), vol. 2318. Singapore: Springer, 2025, pp. 78–92, doi: 10.1007/978-981-97-9793-6_6.
[6] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3Test: Assertion augmented automated test case generation," Inf. Softw. Technol., vol. 176, art. 107565, Dec. 2024.
[7] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in Proc. 26th Symp. Operating Syst. Princ., 2017, pp. 1–18.
[8] C. Martínez-Fernández et al., "Software engineering for AI-based systems: A survey," ACM Trans. Softw. Eng. Methodol., 2022.
[9] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," IEEE Trans. Softw. Eng., 2020.
[10] A. Faraji and N. Pombo, "AI-driven software test automation: An AI4SE-oriented survey of techniques, tools, and challenges," IEEE Access, vol. 13, pp. 183296–183313, 2025, doi: 10.1109/ACCESS.2025.3623944.
[11] M. Borg, C. Englund, K. Wnuk, et al., "Safely entering the deep: A review of verification and validation for machine learning and a challenge elicitation in the automotive industry," J. Automot. Softw. Eng., vol. 1, pp. 1–19, 2020, doi: 10.2991/jase.d.190131.001.
[12] F. Ricca, B. Garcia, M. Nass, and M. Harman, "Next-generation software testing: AI-powered test automation," IEEE Software, vol. 42, no. 4, pp. 25–33, Jul.–Aug. 2025, doi: 10.1109/MS.2025.3559194.
[13] G. Fraser and A. Arcuri, "Whole test suite generation," IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276–291, Feb. 2013, doi: 10.1109/TSE.2012.14.
[14] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using EvoSuite," ACM Trans. Softw. Eng. Methodol., vol. 24, no. 2, art. 8, Dec. 2014, doi: 10.1145/2685612.
[15] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," Empirical Softw. Eng., vol. 20, no. 3, pp. 783–812, Jun. 2015.
[16] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in Companion to the 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems and Applications (OOPSLA '07), New York, NY, USA, 2007, pp. 815–816, doi: 10.1145/1297846.1297902.
[17] G. Wang, Q. Xu, L. Briand, and K. Liu, "Mutation-guided unit test generation with a large language model," IEEE Trans. Softw. Eng., vol. 52, no. 5, pp. 1657–1671, May 2026, doi: 10.1109/TSE.2026.3682975.
[18] C. Kumar, U. Sri Ponaka, P. V. L. N. Naidu, P. Bhuvaneswari, M. Prasad, and S. K. Singh, "AI-powered unit test generation via multi-LLM chaining: A case study with GPT-4o, Gemini, and Claude-3.5," IEEE Access, vol. 13, pp. 204058–204071, 2025, doi: 10.1109/ACCESS.2025.3637221.
[19] S. Karpurapu et al., "Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation," IEEE Access, vol. 12, pp. 58715–58721, 2024, doi: 10.1109/ACCESS.2024.3391815.
[20] A. Mathur, S. Pradhan, P. Soni, D. Patel, and R. Regunathan, "Automated test case generation using T5 and GPT-3," in Proc. 9th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS), Coimbatore, India, Mar. 2023, pp. 1986–1992, doi: 10.1109/ICACCS57279.2023.10112971.
[21] M. Lafi, T. Alrawashed, and A. M. Hammad, "Automated test cases generation from requirements specification," in Proc. Int. Conf. Inf. Technol. (ICIT), Amman, Jordan, Jul. 2021, pp. 852–857, doi: 10.1109/ICIT52682.2021.9491761.
[22] A. Ahmad and M. R. Naeem, "Boosting test generation in industrial codebases: A comparative study of base and fine-tuned LLMs," IEEE Access, vol. 14, pp. 73436–73454, 2026, doi: 10.1109/ACCESS.2026.3690571.
[23] S. Gu, Q. Zhang, C. Fang, F. Tian, L. Zhu, J. Zhou, and Z. Chen, "TestART: Improving LLM-based unit testing via co-evolution of automated generation and repair iteration," arXiv:2408.03095, 2024.
[24] A. Gu, N. Jain, W.-D. Li, M. Shetty, Y. Shao, Z. Li, D. Yang, K. Ellis, K. Sen, and A. Solar-Lezama, "Challenges and paths towards AI for software engineering," arXiv:2503.22625, 2025.
[25] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," IEEE Trans. Softw. Eng., vol. 50, no. 1, pp. 85–105, Jan. 2024.

[26] E. M. Pasca, D. Delinschi, R. Erdei, and O. Matei, "LLM-driven, self-improving framework for security test automation: Leveraging Karate DSL for augmented API resilience," IEEE Access, vol. 13, pp. 56861–56886, 2025, doi: 10.1109/ACCESS.2025.3554960.
[27] C. Wang, F. Pastore, A. Goknil, and L. C. Briand, "Automatic generation of acceptance test cases from use case specifications: An NLP-based approach," IEEE Trans. Softw. Eng., vol. 48, no. 2, pp. 585–616, Feb. 2022, doi: 10.1109/TSE.2020.2998503.
[28] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," IEEE Trans. Softw. Eng., vol. 36, no. 2, pp. 226–247, Mar./Apr. 2010.
[29] P. Delgado-Pérez, A. Ramírez, K. J. Valle-Gómez, I. Medina-Bulo, and J. R. Romero, "InterEvo-TR: Interactive evolutionary test generation with readability assessment," IEEE Trans. Softw. Eng., vol. 49, no. 4, pp. 2580–2596, Apr. 2023.
[30] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), Silicon Valley, CA, USA, Nov. 2013, pp. 246–256.
[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in Proc. 29th Int. Conf. Softw. Eng. (ICSE), Minneapolis, MN, USA, May 2007, pp. 75–84.
[32] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A lightweight UI-guided test input generator for Android," in Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion (ICSE-C), 2017, pp. 23–26.
[33] Z. Liu et al., "Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions," 2023, doi: 10.48550/arXiv.2310.15780.
[34] T. Su, J. Wang, and Z. Su, "Benchmarking automated GUI testing for Android against real-world bugs," in Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), Athens, Greece, Aug. 2021, pp. 119–130.
[35] M. R. Arya Devi and P. Abdul Jabbar, "Automated test case generation using natural language processing," in ICT for Intelligent Systems (ICTIS 2025), Lecture Notes in Networks and Systems, vol. 1519. Singapore: Springer, 2026, doi: 10.1007/978-981-96-8901-9_12.
[36] A. Jabbar and S. Subramani, "Software metrics enhance test data generation and productivity measurement," 2011 World Congress on Information and Communication Technologies, Mumbai, India, 2011, pp. 116-119, doi: 10.1109/WICT.2011.6141228.