How does LLM-based test generation for Java libraries perform when full source code is available?

Evaluating LLM-based test generation for libraries across code representations

Cao Minh Nguyen[1]

Supervisor(s): Sebastian Proksch[1]**, Cathrine Paulsen**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Cao Minh Nguyen
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Cathrine Paulsen, Soham Chakraborty

Abstract

As software projects depend heavily on open-source libraries, developers use tests to ensure that dependency updates remain behaviourally compatible. However, such library tests are often incomplete or unavailable. Although automated test generation tools such as EvoSuite exist and Large Language Models (LLMs) have shown promise in generating more readable tests, most evaluations have been conducted on benchmark datasets or popular GitHub projects. This creates a gap in understanding how effective LLM-generated tests are for released library artifacts. In this paper, we evaluate LLM-based test generation for released Java libraries from Maven Central to assess its feasibility in dependency validation workflows. We implement a pipeline that provides source code and method context to a locally hosted LLM, validates generated tests, and applies iterative repair when needed. Our results show that tests generated by the local model achieve substantially lower coverage than EvoSuite, primarily due to compilation failures, highlighting that symbol resolution errors remain a key challenge in generating tests with LLMs. We further show that iterative repair is effective at improving the coverage of generated tests and a stronger cloud-hosted model even surpasses EvoSuite in coverage. Overall, the findings indicate that LLM-based test generation from source code is a promising approach for dependency update validation when combined with sufficiently capable models and iterative repair mechanisms.

1 Introduction

Modern software systems heavily rely on open-source libraries, which account for 70–90% of the code in typical software projects [2]. Although this accelerates development, it introduces maintenance challenges as dependencies must be regularly updated for bug fixes, performance improvements, and security patches [11]. These updates can also introduce breaking changes in the client software [17]. Although automated dependency management tools such as Dependabot assist developers by proposing updates, they do not guarantee behavioural compatibility [11]. Consequently, validating dependency updates through testing remains challenging, as client tests often provide incomplete coverage while library test suites are frequently unavailable [12; 21].

Automated unit test generation has long been proposed to reduce the manual effort developers need to validate these updates. Search-based software testing (SBST) tools such as EvoSuite generate unit tests by employing search and optimization algorithms to maximize code coverage objectives [5]. Although achieving reasonable coverage, tests generated by EvoSuite are often large, redundant, and less readable than manually written tests, making it difficult for developers to understand and use in practice [4; 7; 19; 20]. More recently, LLMs have shown promising results for test generation from source code context, and it can also produce tests that are more readable and closer in style to developer-written tests [1; 24; 25; 26]. However, existing evaluations on LLM-generated tests are largely limited to benchmark datasets or curated GitHub repositories rather than real-world library artifacts.

This creates a gap in understanding how LLM-based test generation performs in realistic dependency management settings, where libraries are consumed as released artifacts rather than curated repositories. This is a practical limitation as without reliable automatically generated tests for libraries, developers would need significant effort in validating dependency updates which increases the risk of breaking changes.

In this paper, we investigate how effective LLM-based test generation is for released Java libraries from Maven Central when full source code is available. We therefore study the following research questions:

RQ1: How does the coverage achieved by LLM-generated tests compare with EvoSuite-generated tests?

RQ2: What proportion of LLM-generated tests compile and execute successfully, and what are the main causes of invalid tests?

RQ3: How does iterative repair affect the validity, coverage, and computational cost of LLM-generated tests?

RQ4: How does pipeline performance differ between a local open-source LLM and a more capable cloud-hosted LLM?

Our goal is to explore whether LLMs can realistically support dependency validation workflows by generating executable and meaningful tests at scale. To investigate this, we design a pipeline that extracts source code and API context from each class of a Java library, generates JUnit tests using an LLM, executes and validates tests with Maven, and applies an iterative repair loop to fix invalid tests.

The main findings of our study are: (RQ1) the local Qwen2.5-Coder-7B model underperformed EvoSuite in coverage; (RQ2) this was mainly due to compilation failures with unresolved symbols the most common cause; (RQ3) iterative repair improved validity and coverage up to two iterations; and (RQ4) the stronger DeepSeek V4 Flash model substantially improved test quality, surpassing EvoSuite in coverage.

This study makes the following contributions:

- An open-source pipeline for generating and evaluating LLM-generated tests for released Java libraries.
- An analysis of coverage, validity, iterative repair effects, and computational cost of LLM-generated tests.
- A source code baseline for future comparisons with LLM-generated tests using different input settings (e.g., bytecode or bytecode with documentation).

2 Related Work

2.1 Existing LLMs vs. SBST Studies

EvoSuite is one of the most widely used SBST tools for automated unit test generation. It generates Java unit tests by evolving whole test suites towards structural coverage goals and adding assertions based on observed program behaviour

[5; 6]. Existing comparative studies reported mixed results when comparing LLM-based test generation with EvoSuite. Several studies found that LLM-generated tests achieve lower line and branch coverage, often due to compilation failures [1; 25; 26]. In contrast, other work has shown coverage exceeding EvoSuite when including repair loops, feedback mechanisms, or specialised prompting strategies [3; 9; 28; 29]. These mixed results indicate that quality of LLM-generated test suites is highly sensitive to context provisioning, prompting strategy, and repair mechanisms.

For example, Abdullin et al. compared LLM-based test generation with EvoSuite on the GitBug Java dataset, showing that LLM-generated tests achieved lower structural coverage than SBST and symbolic execution, but higher mutation scores [1]. Similarly, Tang et al. compared ChatGPT with EvoSuite for unit test generation and reported that EvoSuite generally achieved higher code coverage, whereas ChatGPT produced tests that were more readable and closer to human-written tests [26]. These findings suggest that LLM-based and SBST approaches show different strengths in terms of coverage versus test quality and readability.

2.2 LLM-Based Unit Test Generation

Several studies have evaluated LLMs for generating unit tests from source code and rich contextual information. Schäfer et al. introduced TestPilot, which generates tests for JavaScript npm packages using function implementations and documentation examples as context [24]. Their evaluation is relevant because it considered real-world library packages.

For Java, Siddiq et al. evaluated LLM-generated unit tests on the HumanEval and EvoSuite SF110 datasets using compilation, correctness, coverage, and test smell metrics [25]. Their results highlighted the difficulty of generating valid tests for strongly typed Java programs and the need for post-processing model outputs. However, their evaluation was conducted on benchmark datasets rather than released Maven library artifacts.

Several approaches have improved LLM-based unit test generation by integrating program context with validation and repair mechanisms. ChatUniTest incorporates adaptive focal context to provide relevant class- and method-level information within prompt limits, followed by syntactic, compilation, and runtime validation with iterative LLM-based repair [3]. ChatTester decomposes generation into intention understanding and test construction, refining non-compiling tests using compiler feedback and additional code context [29]. TestART extends this idea by combining template-based repair of compilation and runtime errors with coverage-guided feedback to generate additional tests for uncovered behaviour [9]. These studies collectively motivate our pipeline design, which uses extracted API context, validation-guided iterative repair, and evaluation using code coverage.

2.3 Knowledge Gap

Most existing studies on the performance of LLM-based test generation for Java, as well as comparisons with EvoSuite, evaluated their approaches on benchmark datasets such as HumanEval, EvoSuite SF110 [25], Defects4J [16; 26], and GitBug [1], or on popular GitHub projects [3]. Although these popular benchmarks allow direct comparisons across different studies, they do not reflect how developers typically interact with software libraries in practice, where dependencies are used as released artifacts from platforms like Maven Central. While TestPilot [24] was evaluated on real package distributions, its evaluation focused on JavaScript npm packages rather than Java libraries.

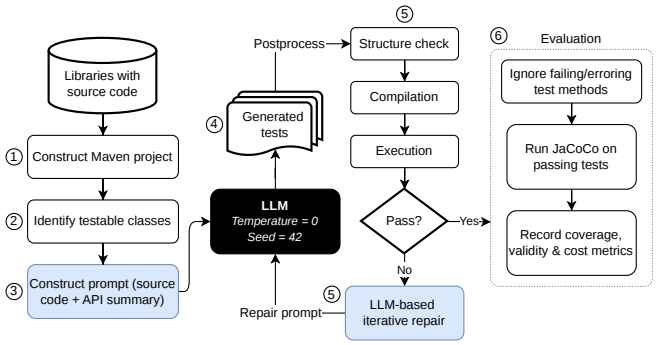

Figure 1: Overview of the implemented pipeline for LLM-based test generation and evaluation.

This paper addresses this gap by evaluating test generation using LLMs on released Java library artifacts from Maven Central with available source code, extending prior evaluations beyond benchmarks and curated projects.

3 Methodology

We implemented a pipeline in Python to evaluate the effectiveness of LLM-generated unit tests for Java libraries. Figure 1 presents the overall workflow, while the implementation and reproduction details are available on GitHub.[1]

3.1 Step 1: Library Preparation

Initially, we obtained the top 1000 most popular Java libraries by querying libraries.io. This set was then filtered to keep only libraries available on Maven Central Repository (MCR) and had version-level popularity metrics available on deps.dev, resulting in 930 libraries. For each remaining library, we selected a representative release based on recency and popularity by choosing the most popular version within the latest major release at the time of collection (specified by `evaluated_at`). Each selected library is uniquely identified by its Maven coordinate (group ID, artifact ID, version).

These 930 libraries were further filtered to include only those (i) present in the EvoSuite result set to ensure comparability with the baseline, and (ii) containing fewer than 40 testable classes (defined in Step 2) to limit runtime per library and allow a broader set of libraries to be evaluated. We refer to this filtering result as dataset A. From dataset A, we randomly selected 20 libraries for RQ3 to determine the optimal number of repair attempts. After this, an additional 14 libraries were randomly selected from dataset A to form the final set of 34 libraries used in RQ1, RQ2, and RQ4. As evaluating multiple repair attempts is computationally expensive, RQ3 was conducted on 20 libraries to ensure feasibility

[1] https://github.com/caominhnguyen05/llm-test-generation

within the project timeline, which may limit the generalisability of its findings.

For each sampled library, the pipeline downloads its binary JAR and `-sources.jar` from Maven Central and creates a Maven test project containing JUnit 4.13.2 and Mockito 4.11.0. Generated tests are later executed against the binary JAR, which is also used by JaCoCo for coverage measurement. The source JAR is extracted into a `prompt_sources` folder for LLM prompting only and is not compiled.

3.2 Step 2: Source File Selection

To reduce computational cost and execution time, the pipeline uses pattern matching to filter out classes that are unlikely to produce meaningful tests. The pipeline excludes the following files: `package-info.java` and `module-info.java`, files without a matching top-level declaration, annotation types, interfaces without executable `default` or `static` methods, enums containing constants only, and classes without detectable method or constructor bodies. These criteria are motivated by prior LLM-based test generation studies which define the generation target as an executable function, method, or class under test, and construct prompts around the focal code implementation [3; 24; 25].

3.3 Step 3: API Context Extraction

Prior work on LLM-based test generation highlights the importance of providing relevant focal context in the prompt. For example, ChatUniTest parses Java classes into Abstract Syntax Trees and extracts class- and method-level information, including imports, fields, method signatures, and method bodies, to support test generation [3]. Motivated by this approach, our pipeline constructs a compact API summary for each focal class and includes it in the prompt alongside the source code. To generate this summary, the pipeline uses JavaParser [15] to parse each Java class and record the signatures of its public constructors and methods. This produces a lightweight representation of the class's callable API.

Figure 2 illustrates an example of the extracted API summary and how it is incorporated into the LLM prompt.

3.4 Step 4: Test Generation & Postprocessing

LLM Setup

The main experiments use the locally hosted Qwen2.5-Coder-7B model [13] through Ollama. This model was selected after preliminary trials with several code-oriented models available in the hardware environment, including CodeLlama-7B, DeepSeek-Coder-6.7B, and Qwen3.5-9B. Among these models, Qwen2.5-Coder-7B provided the most practical balance between test quality, generation time, and memory usage.

To ensure experimental results are reproducible, the model's temperature is set to 0 [24; 25] and its random seed to 42. Additionally, we selected a context window of 6000 tokens because it was the largest size tested that avoided CPU offloading on the available 6 GB GPU hardware. Each LLM call is limited to a generation timeout of 360 seconds to prevent long generations from blocking the experiment.

Example LLM Prompt for Test Generation (class `JSONArray`)

Generate a complete, compilable JUnit 4 test class for the following Java source code.
Target: org.json.JSONArray

Requirements:
- Output only Java code in a single "'java block
- Use only APIs present in the provided source context
- Every test method must use `@Test` annotation & include explicit assertions
- Focus on boundary cases, branches, and exceptions
- Do not test private methods

API Summary (truncated):

- **Constructors**
  - JSONArray()
  - JSONArray(String)
  - JSONArray(Object)
- **Methods**
  - Object get(int)
  - int getInt(int)

Source Code (truncated):

```
package org.json;
public class JSONArray implements { ... } {
    public JSONArray() { ... }
    public Object get(int index) { ... }
    public int getInt(int index) { ... }
}
```

Figure 2: Truncated example of the LLM prompt used for initial test generation, including the extracted API context.

System Prompt for LLM-Based Test Generation and Repair

You are generating JUnit 4 tests for an existing Maven project.

Generate deterministic, compiling, passing tests using only behaviour and APIs supported by the provided source context. Do not invent members, dependencies, or expected behaviour.

Output only one complete Java test file inside a single "'java code block. Do not include explanations or analysis outside the code block.

Figure 3: System prompt used for LLM-based test generation and repair in the pipeline.

Prompt Design

Each LLM call consists of a shared system prompt and a task-specific user prompt. The system prompt is shown in Figure 3. It defines high-level constraints for test generation, including output format requirements and behavioural grounding, and is inspired by the TestART approach [9].

For initial test generation, the user prompt follows the structure in Figure 2. It combines the class name, package name, extracted API summary, and source code to form the input to the model. The prompt requests JUnit 4 tests to match the format of the EvoSuite baseline, and explicitly instructs the model to restrict generation to the provided API context to reduce hallucination.

When validation identifies a compilation or runtime failure, the repair prompt reuses the focal class context and additionally includes the previously generated test code and Maven error output. The repair priorities are to produce a compil-

ing test class, obtain passing tests, preserve valid existing tests, and retain or improve meaningful behavioural coverage where safely possible.

This design is informed by prior LLM-based test generation studies, which emphasise the importance of relevant code context as input [3; 24; 29]. The validation-guided repair process is motivated by ChatTester's iterative prompt refiner strategy, which uses compiler feedback and additional code context to repair non-compiling tests [29].

Postprocessing of Generated Tests

Each raw LLM response is normalised before validation to reduce syntax and compilation failures, following prior practices that postprocess model outputs before executing tests [3; 10]. Normalisation first extracts the Java test class from the model output, then corrects its package declaration and class name to match that of the class under test. To maximise compilation success, it also inserts required JUnit 4 imports, copies imports from the focal source file, and automatically adds imports for recognised Java utility classes used in the generated code. Finally, each generated test method is assigned a `@Test(timeout = 2000)` timeout to prevent non-terminating tests from blocking the library validation process.

3.5 Step 5: Validation and Iterative Repair

For each class under test (CUT), the pipeline evaluates the generated test class through a generation–validation–repair loop. The initial output is repaired up to the configured maximum number of repair attempts.

Each generated test class is evaluated through three stages:

1. **Structural validation.** The pipeline first checks whether the post-processed LLM output contains `public class <ClassName>Test` and at least one `@Test` annotation. Outputs failing this check are rejected and the pipeline proceeds with the next class.
2. **Compilation validation.** If a generated test class is structurally valid, the pipeline stores the class under `src/test/java` and compiles it with Maven.
3. **Runtime validation.** If compilation succeeds, Maven executes the test class. The class is considered successful only if all tests pass; otherwise assertion failures and runtime errors trigger iterative repair if attempts remain.

When validation fails and repair attempts remain, the pipeline constructs a repair prompt using the Maven error message. The repaired output is then post-processed and re-evaluated through the same three validation stages. This approach follows iterative repair mechanisms used in popular LLM-based test generation frameworks including ChatUniTest [3], TestSpark [23], ChatTester [29], and TestPilot [24].

Tests that still contain compilation errors after the last repair attempt are deleted to avoid breaking the final coverage run [10], while the error messages are recorded for analysis in RQ2. Test classes that compile but still fail at runtime are kept for validity analysis, with their failing methods ignored before measuring coverage.

3.6 Step 6: Evaluation

Once generation and repair are complete for all testable classes in a library, the pipeline evaluates the tests across four dimensions: test coverage, test validity, compilation failure categories, and computational cost.

Coverage Measurement

After all test classes are saved, the pipeline annotates failing and erroring test methods with JUnit's `@Ignore` to exclude them from coverage measurement. The remaining tests are executed with JaCoCo [14] to compute the library's code coverage. We ignore failing methods to prevent invalid tests from inflating the reported coverage. An alternative approach considered was to remove failing test methods entirely, but this was less reliable because it occasionally introduced syntax errors. Another option is to discard any test class containing at least one failing test, but this would also remove valid passing tests and potentially underestimate the coverage. Ignoring only failing methods preserves valid tests while excluding invalid behaviour.

The main coverage metrics reported are line and branch coverage as these are commonly used for evaluating LLM-generated tests [1; 22; 24; 28]. For comprehensive comparison with the EvoSuite baseline, we also record instruction, complexity, method and class coverage provided by JaCoCo.

Validity Metrics

Prior work on LLM-based test generation commonly uses compilation status and runtime outcomes as test correctness metrics [9; 24; 25]. Following this, test validity is measured at both the class and method levels. Let N_{CUT} be the number of classes under test and N_{compiled} the number of generated test classes that compile after repair. The compilation success rate is defined as: $N_{\text{compiled}}/N_{\text{CUT}}$.

Let N_{run} be the number of generated test methods, and N_{pass}, N_{fail}, and N_{error} denote passing, assertion failures, and runtime errors, respectively. We report Runtime Success Rate (RSR), Assertion Failure Rate (AFR), and Runtime Error Rate (RER) as:

$$\text{RSR} = \frac{N_{\text{pass}}}{N_{\text{run}}}, \quad \text{AFR} = \frac{N_{\text{fail}}}{N_{\text{run}}}, \quad \text{RER} = \frac{N_{\text{error}}}{N_{\text{run}}}$$

Other Evaluation Metrics

Test classes that still fail to compile after the maximum number of repair attempts have their final Maven error recorded for compilation analysis in RQ2. Compilation failures are categorised from Maven error messages using rule-based pattern matching, following the error breakdown methodology of ChatTester [29]. Recorded categories include unresolved symbols, signature mismatches, missing imports, access violations, and structural errors.

For RQ3 and RQ4, computational cost is calculated as the average per class under test. The pipeline records the number of LLM calls, repair calls, tokens consumed (prompt and output), LLM generation time, and total runtime for each CUT.

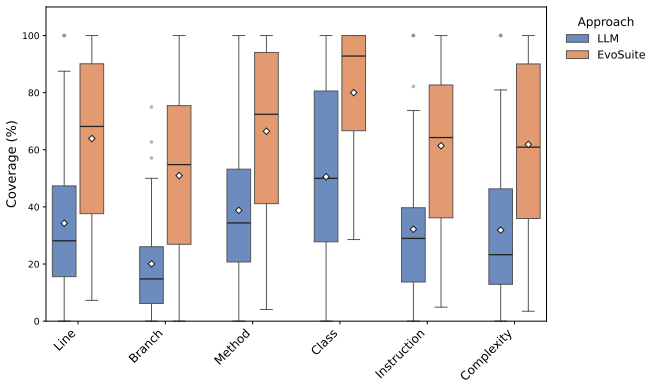

Figure 4: Distribution of coverage achieved by LLM-generated tests and EvoSuite-generated tests across 34 libraries

Table 1: Mean and median coverage percentages achieved by LLM-generated tests and EvoSuite-generated tests across 34 libraries

	LLM		EvoSuite	
Coverage Metric	**Mean**	**Median**	**Mean**	**Median**
Line	34.28	28.14	63.92	68.17
Branch	20.06	14.78	50.95	54.81
Method	38.78	34.38	66.49	72.42
Class	50.57	50.00	80.01	92.82
Instruction	32.18	28.97	61.41	64.28
Complexity	31.87	23.27	61.89	60.92

4 Experimental Results

In this section, we present our results for the four research questions, examining coverage, test validity, the effect of iterative repair, and the influence of model capability on the proposed pipeline.

RQ1: How does the coverage achieved by LLM-generated tests compare with EvoSuite-generated tests?

To assess how well LLMs generate tests for released Java libraries, we ran the pipeline with Qwen2.5-Coder-7B on the 34-library sample defined in Section 3.1. The local LLM's performance was compared with EvoSuite in terms of code coverage, using two repair attempts selected from the results of RQ3. As coverage varies considerably across libraries, median coverage is used as the primary summary statistic. Figure 4 shows the coverage distributions and Table 1 reports the mean and median values of the two approaches.

From Table 1, EvoSuite achieved substantially higher coverage than the local LLM across all six coverage metrics. At the mean level, the largest difference was 30.89 percentage points for branch coverage, where EvoSuite achieved 50.95% compared with 20.06% for the LLM. The largest median difference was in class coverage, where EvoSuite outperformed LLM by a significant 42.82 percentage points. EvoSuite also exceeded the LLM by an identical 40.03 percentage points for both median line and branch coverage. The particularly low median branch coverage achieved by the local model (14.78%) suggests that generated tests have lower complexity and exercise relatively few alternative execution paths.

Figure 4 further illustrates that EvoSuite consistently achieved higher coverage across the sampled libraries. Among the 26 libraries with non-empty line and branch coverage results, the local LLM outperformed EvoSuite on both metrics in only two cases, whereas EvoSuite achieved higher coverage on both metrics in 21 cases. For the LLM, class coverage shows the widest variability across libraries. This may be partly due to compilation failures, as non-compiling test classes were discarded and did not contribute to coverage. More broadly, median method, line, branch, and complexity coverage all remained below 35%, indicating that LLM-generated tests explored only a limited portion of each library's functionality.

Table 2: Validation outcomes of LLM-generated tests by Qwen2.5-Coder-7B across 34 sample libraries

Metric	**Count**	**Percentage**
Test-class level		
Classes under test (testable sources)	350	–
Test classes that compiled	197	56.29%
Test-method level		
Generated test methods	1043	–
Passed test methods	816	78.24%
Assertion failures	191	18.31%
Runtime errors	34	3.26%

RQ2: What proportion of LLM-generated tests compile and execute successfully, and what are the main causes of invalid tests?

To better understand the low coverage observed in RQ1, RQ2 evaluates the validity of LLM-generated tests by measuring compilation and execution outcomes and analysing the causes of compilation failures. These insights inform the design of repair mechanisms and prompting strategies to improve test validity.

We recorded compilation success rate, runtime success rate, assertion failure rate, and runtime error rate of LLM-generated tests, as defined in Step 6. Table 2 summarises these outcomes across the 34 sampled libraries with 350 classes under test.

Table 2 indicates that compilation was a major limitation of the local LLM. Of the 350 CUTs, only 197 generated test classes compiled successfully (56.29% compilation success rate). Since non-compiling test classes were removed before recording coverage, these failures prevented all test methods in those classes from contributing to coverage. This helps explain the lower coverage achieved by the LLM in RQ1, as a substantial proportion of the generated test suite could not exercise the target library.

Among the 1043 generated test methods in test classes that compiled, 816 passed (78.24% runtime success rate), while assertion failures (18.31%) were much more common than runtime errors (3.26%). Two additional test methods (0.19%) were skipped during execution. In general, compilation failures prevented a large number of LLM-generated

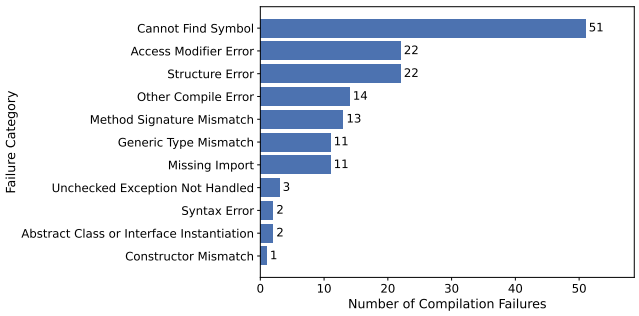

Figure 5: Compilation failure categories of LLM-generated test classes across 34 sample libraries using Qwen2.5-Coder-7B with two repair attempts

test classes from being executed, while incorrect assertions were the dominant problem among executable tests.

To further investigate why more than 40% of test classes failed to compile, Figure 5 summarises the causes of the non-compiling test classes.

The most frequent failure category was *cannot find symbol*, accounting for 51 cases (33.55%). This was followed by *access modifier error* and *structure error*, each occurring in 22 cases (14.47%). Access modifier errors occurred when generated tests attempted to access non-public members of the target class. Structural errors occurred when the model response did not conform to a valid Java test class format, most commonly because it returned explanatory text or refused the request. This may partly reflect the local model's limited 6000-token context size when processing larger focal classes. The remaining failures primarily consisted of method signature mismatches and generic type mismatches, indicating that the local model often struggled to correctly interpret and use library APIs.

RQ3: How does iterative repair affect the validity, coverage, and computational cost of LLM-generated tests?

Many LLM-based testing frameworks use iterative repair mechanisms where test errors are fed back to the LLM to improve test correctness [3; 23; 24; 29]. Although this approach is widely adopted, few studies have analysed how test effectiveness changes with different numbers of repair iterations or how many repair attempts are needed before diminishing returns occur. Motivated by this gap and the low coverage seen in RQ1, RQ3 evaluates the effect of iterative repair on the quality of LLM-generated tests and provides preliminary evidence on an optimal maximum number of repair attempts.

The pipeline was executed with Qwen2.5-Coder-7B on 20 libraries with 214 CUTs, using maximum repair budgets of 0, 1, 2, and 3 attempts. Table 3 reports validity, median line and branch coverage, and average computational cost per CUT for each repair budget.

From Table 3, iterative repair considerably improves the compilation success rate of LLM-generated tests, increasing from 43.93% with no repair to 58.41% at two attempts. Runtime success rate also improves steadily across repair budgets, though much more modestly, increasing from 73.32% to 77.57% over the same range. Overall, validation feedback corrected a large proportion of invalid tests, although benefits decreased with additional repair attempts while incurring additional computational cost.

Table 3: Summary of validity, coverage, and computational cost under different maximum numbers of repair attempts across 20 libraries and 214 classes under test

Metric	**0**	**1**	**2**	**3**
Validity				
Compiled test classes	94	113	125	124
Compilation success rate (%)	43.93	52.80	58.41	57.94
Runtime success rate (%)	73.32	76.60	77.57	78.58
Coverage				
Median line coverage (%)	22.30	30.32	34.57	33.40
Median branch coverage (%)	12.84	19.63	22.53	19.50
Computational cost per class under test				
LLM calls/class	1	1.7	2.27	2.82
Tokens/class	2488.6	5145.3	6904.2	8900.5
LLM generation time/class (s)	15.98	31.45	42.74	54.40
Pipeline runtime/class (s)	23.05	41.87	55.63	68.98

Repair iterations also improve test coverage: median line coverage increases from 22.30% to 34.57% and median branch coverage from 12.84% to 22.53% when moving from zero to two repair attempts. However, these gains come at considerable cost as token consumption nearly triples (2,488.6 to 6,904.2) and pipeline runtime more than doubles (23.05s to 55.63s).

Notably, increasing the repair budget from two to three attempts results in diminishing returns. Pipeline runtime increases by 13.35s and token consumption grows by nearly 2,000 tokens per CUT, yet median line and branch coverage decreases by 1.17 and 3.03 percentage points respectively, indicating that additional repair iterations can occasionally discard previously valid tests. Compilation success rate also decreases slightly. Consequently, two repair attempts provide the best trade-off between validity, coverage, and computational overhead, and were selected as the repair budget for RQ1, RQ2, and RQ4.

RQ4: How does pipeline performance differ between a local open-source LLM and a more capable cloud-hosted LLM?

Prior work has shown that model choice can significantly influence test generation quality, with more capable models often producing syntactically correct tests with higher coverage [1; 8; 25]. Motivated by the limited test validity and coverage observed for the local model in RQ1-RQ3, RQ4 investigates whether these limitations are primarily caused by model capability. We therefore provide an exploratory comparison between the local Qwen2.5-Coder-7B model and a stronger cloud-hosted model, DeepSeek V4 Flash via OpenRouter. DeepSeek V4 Flash is a mixture-of-experts model with 13 billion activated parameters and a 1 million-token context window [18]. The experiment used the same 34 libraries and 350 CUTs as RQ1 and RQ2, with two repair at-

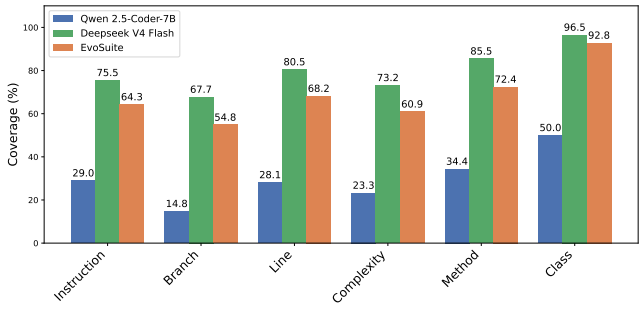

Figure 6: Median coverage achieved by the proposed pipeline using Qwen2.5-Coder-7B and DeepSeek V4 Flash with two repair attempts, with EvoSuite included as a baseline, across the 34 sample libraries.

Table 4: Validity and computational cost of the proposed pipeline using Qwen2.5-Coder-7B and DeepSeek V4 Flash across 34 libraries and 350 CUTs, with two repair attempts

Metric	**Qwen2.5**	**DeepSeek**
Test validity		
Compiled test classes	197	313
Compilation success rate	56.29%	89.43%
Executed test methods	1043	5245
Passed method rate	78.24%	98.47%
Computational cost per class under test		
LLM calls/class	2.34	1.85
Tokens/class	6778.97	11825.39
LLM generation time/class (s)	47.26	67.39
Pipeline runtime/class (s)	62.27	83.50

tempts applied in both settings. Figure 6 and Table 4 present the resulting comparisons in terms of coverage, validity, and computational cost.

Figure 6 shows that DeepSeek V4 Flash substantially outperforms Qwen2.5-Coder-7B across all coverage metrics, with the largest gains in line coverage (80.5% vs. 28.1%) and branch coverage (67.7% vs. 14.8%). Notably, DeepSeek also surpasses the EvoSuite baseline in coverage, improving median line and branch coverage by 12.3 and 12.9 percentage points, respectively.

Table 4 further highlights improvements in test validity and complexity. DeepSeek generates a much higher proportion of compilable test classes (89.43% vs. 56.29%) and produces over five times more test methods than Qwen2.5. This corresponds to 16.76 executed methods per class for DeepSeek, compared with 5.29 for Qwen2.5. Combined with a substantially higher test passing rate (98.47% vs. 78.24%), these results indicate that DeepSeek produces larger test suites with more valid and passing tests.

Regarding cost, DeepSeek requires fewer LLM calls per class than Qwen2.5, indicating that it more often produces valid outputs within fewer repair attempts. However, the gains in test quality come at the cost of 1.74× more tokens and 1.43× longer LLM generation time.

5 Discussion

In this section, we discuss the research and practical implications of our findings regarding test validity, repair strategies, and model capability in LLM-based test generation.

Compilation Failures as the Primary Limitation

A key contribution of our work is identifying test validity as a primary limitation for LLM-based test generation on released libraries. Results of RQ1 and RQ2 suggest that coverage loss was mainly caused by compilation failures rather than an inability to generate potentially useful tests. In particular, unresolved symbols remained the most common compilation failure even when API context was provided in the prompt. This indicates that smaller models struggle to reliably ground generated code in the available API information and instead hallucinate non-existent types and methods. Prior work also reported symbol resolution errors as the primary cause of invalid tests created by LLMs [25; 29], and our results demonstrate that the problem persists in a realistic library setting. Therefore, future research should investigate techniques to improve test validity and API understanding, as reducing compilation failures appears to be the most effective way to achieve higher coverage.

The prevalence of compilation failures also motivates future work to use method-level test generation, as explored by SymPrompt [22] and AthenaTest [27], where individual methods are generated and repaired independently rather than producing a complete test class in a single step. Although this approach increases the number of LLM calls and associated costs [1], it may reduce the impact of individual test errors and improve overall test validity.

Benefits and Limits of Iterative Repair

Our findings in RQ3 provide preliminary evidence that iterative repair is effective but exhibits diminishing returns. Coverage and compilation success improve substantially from zero to two repair attempts, while a third repair increases computational cost but reduces coverage slightly. This demonstrates that more repair iterations are not always beneficial in automated testing pipelines. In particular, two repair attempts provided the best balance between effectiveness and cost, suggesting that future frameworks should use adaptive repair strategies rather than fixed repair budgets, for example by stopping when no improvement in compilation or coverage is observed between iterations to minimise computational cost.

Regarding practical implications, the results suggest that iterative repair should be a standard component of production LLM-based testing systems, as even a single repair step substantially improves the proportion of executable tests and overall coverage. However, the benefits decrease quickly after one iteration, indicating that repair mechanisms should be treated as a lightweight enhancement rather than the primary strategy for ensuring test quality in open-source library settings. This is because repair primarily addresses syntax and compilation errors, while deeper limitations in API understanding and behavioural reasoning remain unresolved. Consequently, practical deployment should combine automated

execution and error feedback with structural improvements to test generation, such as method-level generation for more fine-grained error handling, alongside the use of stronger models with improved API grounding capabilities.

Model Capability Strongly Determines Test Quality

RQ4's results indicate that LLM-based test generation is strongly dependent on model capability. The local Qwen2.5-Coder-7B model, with a 6000-token context window, achieved only 28.1% and 14.8% median line and branch coverage, and in some cases refused to follow instructions on more complex classes, responding with "I'm sorry, but I can't assist with that request.". These failures suggest that increased task difficulty can cause smaller models to fail to follow instructions, leading to incomplete test generation. Overall, model capability affects not only coverage but also the reliability in task completion. This observation is consistent with Abdullin et al., who found that 4–8k context sizes are insufficient for real-world applications [1]. In contrast, the stronger DeepSeek V4 Flash model substantially improved compilation success, while also exceeding EvoSuite in coverage.

This opens directions for future research to study how model size, context length, and prompting strategies each contribute to performance, and whether smaller models can close the gap through techniques such as retrieval augmentation or method-level generation. It also shows the trade-off between cost and effectiveness when comparing SBST methods with more capable LLMs, especially in large-scale testing scenarios, where strong models require significant financial cost for token usage but can achieve higher coverage.

From a practical perspective, the results indicate that modern highly capable LLMs can already serve as viable alternatives or complements to SBST tools such as EvoSuite, especially in dependency validation workflows where readable and maintainable tests are important. However, the weaker performance of the local LLM suggests that smaller open-source models are not yet reliable enough for fully automated use without additional support. Moreover, a promising direction is combining LLMs with EvoSuite, as the two approaches offer complementary strengths. LLMs tend to produce readable, semantically meaningful tests while EvoSuite systematically maximises structural coverage, potentially achieving stronger overall test suites.

Threats to Validity

Several factors may affect the validity of our results.

Construct validity. Our study uses code coverage to measure test effectiveness. While coverage indicates how much code is exercised, it does not capture fault detection capability or assertion quality. Thus, coverage may not fully reflect the practical effectiveness of the generated test suites. To mitigate this, we additionally report runtime success rates and assertion failure rates, which provide complementary signals about test correctness beyond structural coverage.

External validity. We ran the experiments on a sample of 34 libraries, each with a relatively small number of classes. Although the libraries represent real Maven artifacts, the results may not generalise to larger libraries with more complex dependency structures, broader APIs, or different application domains. To partially mitigate this threat, the sample dataset includes libraries from different domains and varying numbers of classes, allowing us to capture a range of real-world usage scenarios.

Model validity. Using two LLMs (one local and one cloud-hosted) enabled a useful comparison of how model capability affects test generation effectiveness. However, the findings may not generalise to other models with different training data, reasoning capabilities, or context sizes.

6 Responsible Research

Reproducibility and data availability. To support reproducibility, the sample datasets, source code, prompt templates, complete experimental results, and execution instructions are publicly available on Zenodo at https://doi.org/10.5281/zenodo.20718478. We set temperature and random seeds for both LLMs to improve determinism across runs, although exact reproducibility cannot be guaranteed for the cloud-hosted model due to its reliance on external providers. The dataset consists of 34 libraries from Maven Central selected to represent a range of domains and project sizes; however, we acknowledge that this selection may introduce bias towards simpler and smaller libraries and may not fully capture the diversity of all Java libraries.

Ethical and safety considerations. The study uses publicly available open-source library artifacts from Maven Central and does not involve human participants or proprietary data. Executing LLM-generated tests introduces risks such as unintended file operations, network access, and non-terminating execution. These risks are mitigated through prompt constraints and execution timeouts, but future use of the pipeline should employ isolated or sandboxed environments to further reduce potential safety issues.

Computational cost and environmental impact. LLM-based test generation incurs considerable computational cost due to repeated model calls, iterative repair, and test execution. In particular, the stronger cloud-hosted model significantly increases token usage and runtime compared to the local model. While we do not explicitly optimise for energy efficiency, we report detailed computational cost metrics to support future research on more sustainable and cost-effective LLM-based testing approaches.

Use of LLMs. LLMs such as Codex and ChatGPT were used to support pipeline implementation, debugging, and code review. All generated code was manually reviewed and modified where necessary before use. The final responsibility for the experimental design, implementation, and reported results remains with the author.

7 Conclusion

While LLM-based test generation has been widely studied, most existing evaluations focus on benchmark datasets rather than released libraries that developers typically work with. This study investigates whether LLMs can effectively generate tests for real-world Java libraries from Maven Central to support dependency update validation, where reliable tests are often missing or incomplete. We propose an automated

pipeline that extracts source code and API context of focal classes, generates unit tests using LLMs, and applies validation and iterative repair.

Our results show that the local model underperformed EvoSuite due to frequent compilation failures, mainly due to unresolved symbols. Although iterative repair improved validity and coverage, its benefits diminished after two iterations and could not fully address API understanding limitations. In contrast, a more capable cloud-hosted model achieved substantially higher coverage, even surpassing EvoSuite.

Overall, our findings indicate that while LLM-based test generation is promising for supporting the validation of real-world Java libraries, its effectiveness strongly depends on model capability and the ability to reliably ground outputs in library APIs. Iterative repair is an effective strategy for improving test quality and should be incorporated into production LLM-based testing systems. This study motivates future work on method-level test generation to better isolate errors and reduce the impact of compilation failures, as well as on integrating LLMs with EvoSuite to leverage the strengths of both approaches.

References

[1] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 221–232. IEEE, 2025.

[2] Hilary Carter, Cara Delia, Tosha Ellison, Colin Eberhardt, Stephen Hendrick, and Philip Holleran. The 2022 state of open source in financial services, December 2022. Foreword by Gabriele Columbro.

[3] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 572–576, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, page 1449–1461. IEEE Press, 2025.

[5] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 416–419. ACM, 2011.

[6] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[7] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 348–351, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Sijia Gu, Noor Nashid, and Ali Mesbah. Llm test generation via iterative hybrid program analysis. *arXiv preprint arXiv:2503.13580*, 2025.

[9] Siqi Gu, Quanjun Zhang, Kecheng Li, Chunrong Fang, Fangyuan Tian, Liuchuan Zhu, Jianyi Zhou, and Zhenyu Chen. TestART: Improving LLM-based unit testing via co-evolution of automated generation and repair iteration. *arXiv preprint arXiv:2408.03095*, 2024.

[10] Vitor Guilherme and Auri Vincenzi. An initial investigation of chatgpt unit test generation capability. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '23, page 15–24, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering*, 49(8):4004–4022, 2023.

[12] Joseph Hejderup and Georgios Gousios. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software*, 183:111097, January 2022.

[13] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[14] JaCoCo Contributors. JaCoCo: Java code coverage library. https://www.eclemma.org/jacoco/. Accessed: May 25, 2026.

[15] JavaParser Contributors. JavaParser: Java parser and Abstract Syntax Tree for Java. https://github.com/javaparser/javaparser, 2026. Accessed: May 28, 2026.

[16] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440. Association for Computing Machinery, 2014.

[17] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study. *Empirical Softw. Engg.*, 27(3), May 2022.

[18] OpenRouter. DeepSeek: DeepSeek V4 Flash. https://openrouter.ai/deepseek/deepseek-v4-flash, 2026. Model specifications and API pricing page. Accessed: May 23, 2026.

[19] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: an empirical study. In *Proceedings of the 9th International*

Workshop on Search-Based Software Testing, SBST '16, page 5–14, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. Automatic test case generation: what if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 130–141, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Cathrine Paulsen and Sebastian Proksch. Marco: Compatible version ranges in maven. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 910–914, 2025.

[22] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.

[23] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. Testspark: Intellij idea's ultimate test generation companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 30–34, New York, NY, USA, 2024. Association for Computing Machinery.

[24] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024.

[25] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 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, page 313–322, New York, NY, USA, 2024. Association for Computing Machinery.

[26] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Trans. Softw. Eng.*, 50(6):1340–1359, June 2024.

[27] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2021.

[28] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1258–1268, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.