# LLMs as Test Generators: A Comparative Benchmarking Study

Esdras Caleb Oliveira Silva
DIMAp
Natal, Rio Grande do Norte, Brazil
esdras.caleb@ufrn.br

Roberta de Souza Coelho
DIMAp
Natal, Rio Grande do Norte, Brazil
roberta.coelho@ufrn.br

Lyrene Fernandes da Silva
DIMAp
Natal, Rio Grande do Norte, Brazil
lyrene.silva@ufrn.br

## ABSTRACT

Automated tests are a key practice adopted by the software industry to verify software quality. However, they are costly to develop and maintain. Recently, the use of LLMs to generate automated tests has been explored as a viable alternative. Ongoing efforts focus on improving generation by providing richer context and post-processing the output to correct errors, ensuring accurate results. However, small-scale open LLMs, capable of running on modest hardware, have received limited attention. This work compares large-scale LLMs (e.g., GPT and Gemini) with small-scale open-source models in terms of the number of tests generated and their quality, measured by the mutation score, the cyclomatic complexity of generated code, and the number of test smells on them. We evaluated 12 small-scale models against 6 large-scale ones and used EvoSuite to establish a baseline for code quality and the number of methods tested. Our results show that some small-scale LLMs perform well in test generation tasks. xLan, Gemma2, and DeepSeekCoder gave the best overall results, producing as many tests as large-scale models, with fewer smells and a better mutation score.

## KEYWORDS

Automated Tests, LLM, Experimental Study, Benchmarking

## 1 Introduction

Automated testing is widely used in the Software Industry to detect introduced faults along releases (regression checking). They are a key practice to support the quality assurance process [44] in continuous delivery projects. However, implementing and maintaining automated tests is a costly task that requires time, effort, and specialized knowledge from the development team.

Tools such as Randoop [29] and EvoSuite [13] were proposed to make automated test generation less costly. Such tools can generate unit tests for Java software based only on the project's source code. Recently, the emergence of Large Language Models (LLMs) has led to the adoption of commercial tools such as ChatGPT to assist in the automated test generation.

LLMs generate tests that offer advantages compared to existing test generation tools, notably in producing more readable code [43], and supporting multiple programming languages. However, a key disadvantage is that certain LLMs generate incorrect and low-quality (smelly) tests [36].

Moreover, even when LLMs generate good tests, there are other problems associated with their use [34]: (i) widely known large-scale models (e.g., ChatGPT, Gemini) usually need to be executed on external hardware, the code is then shared with multiple external parties, making it more prone to information leaks and even being used for training purposes and leaking valuable information [23]; and (ii) these large-scale LLMs consume too much energy to generate code [10].

One solution to both problems is the use of small-scale, specialized models. In this way, the model runs inside a user infrastructure without the need to share the project code with any other part.

An important question is how these small-scale LLMs compare to the larger ones. In this work, we answer this question by making a benchmark of generative models, comparing large-scale models that require external infrastructure like GPT and Gemini (composed of 10 to 30 billion parameters) to small-scale models that can run on modest hardware (composed of 1 to 3 billion parameters). In total, 18 models were compared (Table 1): 6 models with more than 7B parameters were accessed through Web interfaces, and 12 small-scale models made up of no more than 3B parameters. We then compare the quality of the generated test code based on the number of code smells, mutations killed by the tests (mutation score), the cyclomatic complexity, and the number of generated tests without compilation errors. Our results show that small-scale LLMs can achieve mutation scores comparable to, and in some cases better than, large-scale models, while generating an equal quantity of compilable tests, highlighting their potential for efficient test generation on modest hardware.

The structure of this work is as follows. Section 2 reviews previous studies relevant to this research; Section 3 details the methodology, datasets, and tools used; Section 4 analyzes the findings obtained from the experiment; Section 5 discusses potential limitations and biases in the study; Section 6 summarizes key insights and provides directions for future research.

## 2 Related Works

Automated test generation has been an active research area for several years. One of the first tools in this domain is Randoop [29], which uses feedback-directed random testing to generate test cases heuristically. Building on this approach, EvoSuite [13] introduced evolutionary algorithms to optimize the test generation process, improving code coverage and fault detection of the generated tests.

With the proven capability of large language models (LLMs) for code generation [18, 26], several studies have adapted them for automated test generation. The use of LLMs to support the software testing process is discussed in the systematic review by Wang et al. [41], which found that LLMs are primarily used for unit test generation and code correction. We combine these 2 uses in our study to generate correct compilable tests.

One such approach is AthenaTest [40], which extracts method signatures to construct instructions for an LLM to generate unit tests. A3Test [3] improves on AthenaTest by incorporating domain knowledge into prompt generation and applying static analysis to detect and repair errors in generated tests. Schäfer et al. [35] also use the available documentation to support automated test generation through their tool called TestPilot. TestPilot uses GPT 3.5 to generate tests in JavaScript with not only the code but also documentation and code snippets of the function. They also used a simple procedural analysis to repair syntactic errors in the generated test. Like them, we also use cyclomatic complexity to measure the quality of our tests.

The work of Yuan et al. [43] proposes ChatTester, which improves the generation of tests using GPT by defining the context of the method being tested. This is done by asking the LLM to generate a description of the function. The second improvement involves using GPT to correct syntactic errors based on the compiler feedback. By combining these two techniques, they nearly doubled the number of correctly generated tests. In our work, we extended ChatTester to enable the use of different LLMs for test generation and also to calculate the mutation score (i.e., to run the generated tests against mutations). We also record metrics on the number of iterations required and the ability to kill mutations, and compared the results of large-scale models against small-scale ones. This approach and how the experiment was conducted are presented in Section 3.

Liu et al. [22] examined 13 tools aimed at creating tests, of which 11 generated JUnit tests. They used JaCoCo to measure code coverage, PIT to generate mutants and calculate the mutation score, and JavaParser to count the number of test cases. In our work, we also use the JavaParser to examine the number of methods tested, as well as their cyclomatic complexity.

Siddiq et al. [36] benchmark exclusively the following generative models: GPT3.5, Codex, and StarCoder. They were tested against the HumanEval dataset [5] and the SF110 benchmark projects [13], and compared the generated code to EvoSuite. The comparison consisted of whether the code works after a heuristic fix, coverage, and smells. To detect test smells, they used TsDetect, the tool developed by Peruma et al. [31]. The models performed badly in comparison to EvoSuite, having fewer smells for certain types. In our benchmark, we also use TsDetect to determine code quality.

## 3 Methodology

We conducted an experimental study to compare the test generation capabilities of 18 LLMs, of which 6 are large-scale LLMs and 12 are small-scale models capable of running on a personal computer. Our research used Goal-Question-Metric (GQM) to structure this study. The goal is: "To compare small-scale and large-scale LLMs for the purpose of unit test generation". The research questions derived from this goal are as follows.

- **RQ1: How can we compare large- and small-scale models in terms of the number of generated tests?** To answer this RQ, we used the number of methods tested and the number of tests generated per method as a metric to compare small- and large-scale LLMs.
- **RQ2: How can we compare the quality of the tests generated by small-scale LLMs and large-scale LLMs?** To answer this RQ, we considered the number, type, and proportion of test smells, mutation score, and cyclomatic complexity.
- **RQ3: How can we compare small- and large-scale LLMs in terms of the number of iterations needed to generate tests?** To answer this RQ, we considered the average number of correction attempts required to produce a compilable test, that is, how many times the LLM had to be asked to fix compilation errors in the generated code.

The following metrics were selected to address the research questions, reflecting both the properties of the generated tests and the behavior of the test generation process. The metrics were calculated for each LLM considering the seven target projects presented in Table 2. For metrics calculation purposes, all seven projects were considered together and referred to as Systems Under Test (SUTs) in the metrics description.

- **Number of methods being tested:** This metric measures the extent to which the generated tests cover the methods of Systems Under Test (SUTs). - **RQ1**
- **Mean number of tests generated per method:** This metric represents the mean number of test cases generated for each method of Systems Under Test (SUTs). It is obtained by the sum of all generated tests (for all SUTs) divided by the number of methods being tested. - **RQ1**
- **Mutation Score**: This metric is the percentage of generated mutations detected by the generated tests. Calculated by the sum of mutations killed divided by the number of mutations generated for all SUTs. - **RQ2**
- **Mean Cyclomatic Complexity**: This metric is the mean value of the cyclomatic complexity of the generated tests. It is obtained by the sum of the cyclomatic complexity of the generated tests for all SUTs divided by the number of generated tests. - **RQ2**
- **Mean Number of Test Smells per Test:** This metric is the mean value of Smells per generated tests. It is obtained by the sum of smells present in all generated tests (for all SUTs) divided by the number of generated tests. - **RQ2**
- **Estimated Proportion of Tests Smells:** This metric is the percentage of generated tests with Smells, as presented in Section 3.2. - **RQ2**
- **Most Frequent Smell:** This metric is the most frequent type of smell in the generated tests. - **RQ2**
- **Number of Corrections:** This metric is the average number of correction iterations (with the LLM) needed to generate a test that complies and executes. It is obtained by the sum of the number of corrections made for each test generation, divided by the number of generated tests. - **RQ3**

Since each LLM covers a different number of SUT methods and generates a different number of tests per method, we used the mean as the basis for five of the metric evaluations to ensure a fair comparison of the test quality.

### 3.1 Tools

To carry out this study, we used TsDetect and Lizard, and also developed an extended version of ChatTester [43], which we call ChatTesterMut. Each of these is described in detail below.

*3.1.1 ChatTesterMut.* ChatTesterMut is our extended version of ChatTester [43] used in this work. By refining prompt construction and automatically correcting compilation errors through GPT itself, ChatTester significantly enhances test generation, doubling GPT's capacity to produce compilable tests. ChatTester workflow is illustrated in Fig. 1. The system begins by asking GPT to explain

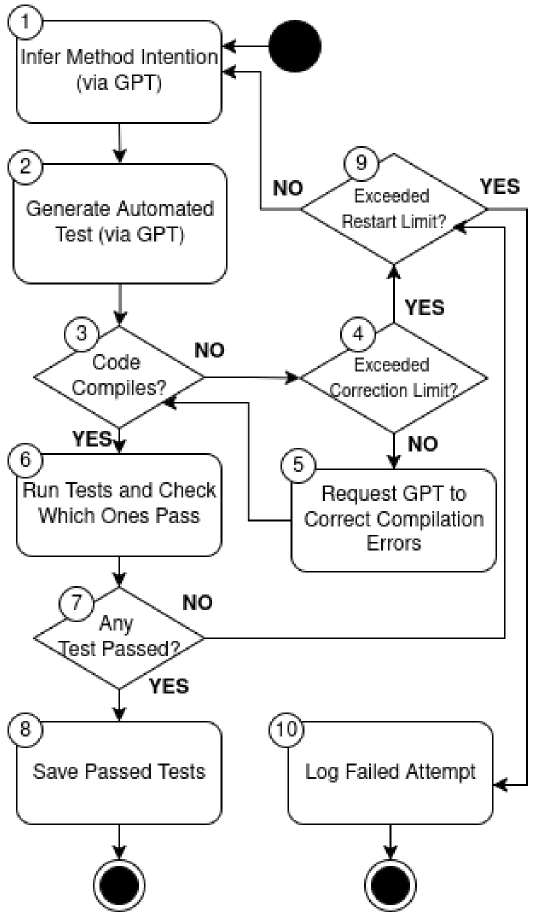
1
Infer Method Intention (via GPT)
2
Generate Automated Test (via GPT)
3
Code Compiles?
NO
4
Exceeded Correction Limit?
YES
9
Exceeded Restart Limit?
NO
YES
NO
5
Request GPT to Correct Compilation Errors
YES
6
Run Tests and Check Which Ones Pass
7
Any Test Passed?
NO
YES
8
Save Passed Tests
10
Log Failed Attempt

Figure 1: ChatTester Workflow

the intent of the method under test (Step 1). Using this explanation along with the class code, it constructs a prompt to generate the test case (Step 2). After generation, the test is checked for compilation errors (Step 3). If a compilation error is found, ChatTester asks GPT to fix it (Step 5). This correction loop continues until the test compiles or a maximum number of retries is reached (Step 4). Once the test compiles, it is executed; if it passes, the test is saved. If no passing test is produced or all compilation attempts fail, ChatTester restarts from Step 1, repeating the process up to five times before giving up.

ChatTesterMut extended ChatTester to support LLMs beyond ChatGPT. In addition to this broader compatibility, it enables the use of six types of mutation operators for mutation testing and records detailed metrics for benchmarking. ChatTesterMut can generate and evaluate tests against the following six mutation types:

(1) **Null Mutation:** This operator converts the value returned by a function to a null-equivalent value. For example, if the function returns a string, it will return an empty string; if it returns an object, it returns `null`; if it returns a number, it will return `0`; and if it returns a Boolean, it will return `false`.
(2) **Variable Mutation:** Given a method call, this operator sends null-equivalent values as its arguments.

~~~
```java
// Focal class
${class_sig} {
        ${fields}
        <#if constructor_sigs?has_content>
                ${constructor_sigs}
        </#if>
        // Focal method
        ${method_body}
}
```
Please infer just the intention of the
method "${method_sig}" in class
"${class_sig}".
Specifying what inputs
"${method_sig}" accepts, what
"${method_sig}" does and what
"${method_sig}" returns, Keep your
response brief.
~~~

Figure 2: Intention Prompt

(3) **Boolean Mutation:** This mutation operator negates a logical expression inside an if clause.
(4) **Arithmetic Mutation:** The arithmetic operator is reversed (i.e., addition to subtraction, division to multiplication, and vice versa).
(5) **Logical Mutation:** The logical operator is inverted in an expression.
(6) **Relational Mutation:** The relational operator is reversed in an expression.

We implement our own mutation solution instead of using existing solutions, such as PIT [7], to enable its integration within ChatTester. Therefore, we incorporated mutation generation directly into the ChatTester workflow. After test generation, a mutated version of the original class is created, and the generated tests are executed against it. If the test fails, the mutation is marked as killed; if it passes, the mutant is considered alive. The mutation score is then calculated as the number of mutants killed divided by the number of mutants generated.

We also refined how the prompts used by ChatTester interact with LLMs. Considering the intention prompt (part 1 in Fig. 1), we explicitly ask what the method does, what input it receives, and what the expected output is. We also included a phrase asking for a brief response in the intention prompt. In doing so, we could obtain intentions smaller than in the original version of ChatTester - sometimes the intention prompt of the original ChatTester was too long and exceeded the prompt limit [21]. Fig. 2 presents the new intention prompt with highlights on improvements. The code listings were formatted using fenced code blocks with Markdown [25] language identifiers.

Furthermore, a context prompt [32] was used to guide the LLM in what we expect from the generated code. This context prompt was used only in the generation and correction prompts, which are parts 2 and 3 of the Fig. 1:

> You are a senior tester in Java projects, your task is writing tests for a specific focal method in a focal class with JUnit5 and Mockito framework (a focal method means a method under test). You need to create a complete unit test using JUnit 5, ensuring to cover all branches. Compile without errors, import the class being tested and use reflection to invoke private methods or fields if needed. Please answer with just the java code.

Moreover, ChatTesterMut implements additional features needed to perform this benchmark study, such as storing the number of iterations with the LLM and calculating the mutation score.

*3.1.2 TsDetect.* Code smells are bad practices that affect code quality (e.g., impairing readability, and increasing the cost of maintenance). The work of Peruma et al. [30] details common types of test smells. We use TsDetect [31] to analyze the generated tests and extract the types of smells in them.

*3.1.3 Lizard.* The cyclomatic complexity of the generated tests was calculated using a Python library called Lizard [27]. It is worth mentioning that Python scripts were also used in conjunction with shell scripts to automate most of the benchmarking process in Fig. 3, reducing the risks of human error throughout the process.

## 3.2 The Benchmark Procedure

The complete benchmark procedure is presented in Fig. 3. First, LLMs were selected. Then we evaluate and select the fit candidates to perform the benchmark. With the selected LLMs in the step 2, we use ChatTesterMut to generate tests for projects 1 to 7 of the SF110 benchmark [13]. ChatTesterMut records the generated tests, mutation scores, and relevant metrics in a file. In step 4, we recover the Evosuit tests for projects 1 to 7 from the SF110 benchmark and compile a similar file with the relevant metrics. TsDetect [31] and Lizard [27] are then used in steps 3 and 5, respectively, to extract complementary metrics on test smells and cyclomatic complexity. These steps result in metrics that will be analyzed to answer our research questions. The following subsections detail each of these steps.

*3.2.1 LLMs Selection.* In step 1 of Fig. 3, we selected the models for the study. To select small-scale LLMs, we use the Hugging Face platform [11], performing individual searches with the keywords: *code*, *1B*, *2B*, *3B*, *small* and *tiny*. For large-scale LLMs, we considered models provided by leading AI companies such as Google, OpenAI, and MistralAI, as well as platforms offering free API access to state-of-the-art large-scale models. After selecting each candidate, the following evaluations were performed:

(1) Ask the model to build Java code for the Fibonacci function.
(2) Check if the model can generate a test for the first method detected in the first project of the SF110 benchmark [13] using the ChatTesterMut.

Then we manually inspect the Fibonacci function and the generated test to see if it makes sense. These evaluations were conducted on an i7 11th generation with 16GB RAM, without a GPU. Models, already mentioned in other works, such as Llama2 or Starcoder, were eliminated at this stage. In Step 2, the maximum CPU usage was 54.9% (and the mean was 52%), the maximum RAM usage was 13GB (and the mean was 6.56GB), and the maximum generation time was 8.3 min (and the mean was 3.6 min).

In this selection, all large-scale LLMs presented in Table 1 passed, although it was necessary to use the free versions of these models due to access limitations to their API. The selection of small-scale LLMs focused on models created and designed to operate at a reduced size. Since their developers had previously evaluated them and validated their viability within this limitation.

Table 1 shows the selected LLMs, their full name, number of parameters, size, where they were executed, and the primary domain they were designed for. The xLan models [45] were created to operate through function calling, which means that the model outputs a function name and its arguments to be executed by an external system. Although this approach does not involve traditional code generation, it has proved to be effective in this task. The bigger models with more than 3 billion parameters were all executed in web providers, and the small-scale models with 3 billion or fewer parameters were executed locally. Some bigger models are closed and do not disclose their size or number of parameters. This is why some are marked with n.d. (not disclosed) and with ~ and >, as their precise values are not available. For the remainder of this work, we refer to the models by the names listed in the Model ID column, Table 1.

*3.2.2 Tests Generation.* In step 2 of Fig. 3, the selected models were used to generate tests for the first seven projects of the SF110 benchmark [13], which are presented in Table 2. The number of methods and lines of code was obtained using Lizard by analyzing the project's *src/main/java* directory, the sent methods refer to the methods detected by ChatTester and sent to LLM.

The generation of tests uses ChatTesterMut, following the steps described in Section 3.1.1. Using ChatTester ensures that all generated tests are correct and pass, avoiding the problems reported by Siddiq et al. [36].

The generation process used the i7 11th gen with 16GB RAM for the web-based large-scale LLMs. Since the previous test with local models on the i7 without a GPU took an average of 3.6 minutes per test, it would have taken a considerable amount of time to generate tests for all 12 local models across all 7 projects. Therefore, we chose to run the small-scale LLM benchmark generation on more powerful hardware, using NPAD/UFRN with an NVIDIA H100 GPU.

After a successful test generation, ChatTesterMut tries to generate mutations to the method of the system under test (SUT). If successful, it checks whether the test kills the mutation. Then it saves the number of mutations created and mutations that the generated tests killed.

*3.2.3 Test Smells Checking.* In step 3, Fig. 3, the tests are then analyzed using TsDetect [31] to check the number and types of test smells. This step is also done with the Evosuit tests.

Since TsDetect only reports the total number of test smells per file without linking them to individual test methods, we can only compute the mean number of smells per test method within each file. Therefore, this mean value was adopted as one of our evaluation metrics. Although TsDetect does not specify which test methods are affected, we can estimate lower and upper bounds on the number of smell-affected methods by analyzing the distribution of smell types across files and the number of test methods per file. Using

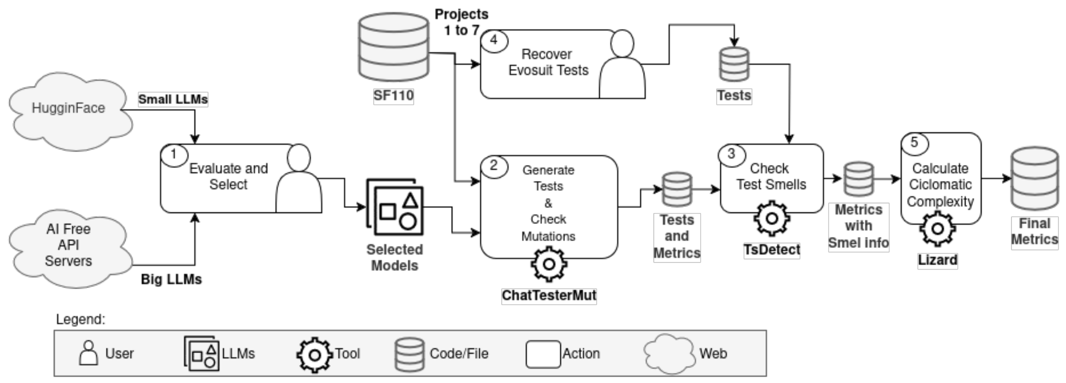

**Figure 3: Test Generation Benchmark Procedure**

**Table 1: Benchmark Models. The symbol * means function calling**

| Model ID | Model Name | Num of Params | Size | Interface | Type |
|---|---|---|---|---|---|
| codestral | codestral-latest [2] | ~24B | n.d. | Web | Code |
| gemini-mini | gemini-1.5-flash-8b [9] | 8B | n.d. | Web | Generalist |
| gemini | gemini-1.5-flash [8] | ~32B | n.d. | Web | Generalist |
| gpt4o | gpt-4o-mini [28] | ~8B | n.d. | Web | Generalist |
| open-codestral | open-codestral-mamba [1] | 7B | 15GB | Web | Code |
| Qwen32B | Qwen2.5-Coder-32B-Instruct [20] | 32B | 65GB | Web | Code |
| Llama3B | Llama-3.2-3B-Instruct [16] | 3B | 6GB | Local | Generalist |
| granite | granite-3.1-1b-a400m-instruct [39] | 1B | 2.7GB | Local | Generalist |
| xLAM | xLAM-1b-fc-r [45] | 1B | 2.7GB | Local | Code* |
| gemma | gemma-2-2b-it [38] | 2B | 5GB | Local | Generalist |
| deepseek | deepseek-coder-1.3b-instruct [17] | 1.3B | 2.7GB | Local | Code |
| Yi | Yi-Coder-1.5B [42] | 1.5B | 3GB | Local | Code |
| Smol | SmolLM2-1.7B-Instruct [4] | 1.7B | 3.4GB | Local | Generalist |
| Qwen1.5B | Qwen2.5-Coder-1.5B-Instruct [20] | 1.5B | 3GB | Local | Code |
| Qwen0.5B | Qwen2.5-Coder-0.5B-Instruct [20] | 0.5B | 1GB | Local | Code |
| OpenCoder | OpenCoder-1.5B-Instruct [19] | 1.5B | 3.8GB | Local | Code |
| Llama1B | Llama-3.2-1B-Instruct [16] | 1B | 2.5GB | Local | Generalist |
| Falcon | Falcon3-1B-Instruct [37] | 1B | 3.3GB | Local | Generalist |

these bounds, we compute an average to approximate how many test methods are likely to be affected by at least one smell, as shown in Fig. 5.

The maximum number of smelly tests is estimated as the smaller value between the total number of smell occurrences and the total number of tests. In other words, a test can contain more than one smell, so the number of smells may exceed the number of tests. Therefore, the maximum number of smelly tests is given by: MIN(total_smells, total_tests).

Among all types of test smells identified by TsDetect [30], the *lazy test* is the only one that requires 2 tests to exist. *Lazy Test* is when two or more tests evaluate the same function. So, if a test file has the *lazy test* smell, it should have at least two tests with this smell, as more than one test needs to call the same system under test (SUT) function. So we have 3 alternatives:

- The test file has no detected Smells, so its minimum number of smells is 0
- The test file has detected Smells, and one of them is *Lazy Test*, the minimum number is 2
- The test file has detected Smells, and none of them is *Lazy Test*, the minimum number is 1

For comparison purposes, every time we refer to tests, we consider the number of functions marked with `@Test` in the file. TsDetect detects smells only in functions marked with `@Test`.

After completion of the smell check (step 3 in Fig. 3), the data from EvoSuite is recovered from SF110 (step 4 in Fig. 3) and submitted to the same test smell check (step 3 again). Then, both the EvoSuite and LLMs data are merged into one file, and the cyclomatic complexity of all test files is calculated (step 5). Section 4 details our findings and the metrics used to answer our research questions.

Section 4 details our findings and the metrics used to answer our research questions.

**Table 2: Projects Data**

| Project | Domain | Type | # Methods | # Sent Methods | LoC |
|---|---|---|---|---|---|
| tullibee | Financing | Middleware | 196 | 137 | 3236 |
| a4j | e-commerce | API Client | 464 | 135 | 3602 |
| gaj | AI | Framework | 47 | 16 | 320 |
| rif | App. Integration | Middleware | 66 | 29 | 953 |
| templateit | Doc. Automation | Template Processor | 177 | 29 | 2463 |
| jnfe | Business | API Backend | 213 | 38 | 2096 |
| sfmis | Education | Management System | 175 | 43 | 1288 |

## 4 Results

Table 3 contains the metrics collected in this study to compare large-scale and small-scale LLMs and to answer the research questions presented in Section 3. Besides collecting and comparing metrics associated with the tests generated by LLMs, we also compare some characteristics of LLMs' generated tests with the ones generated by EvoSuite. Since EvoSuite represents the state of the art in test code generation.

Some metrics, such as mutation score and correction iterations, could not be collected for EvoSuite tests, since they are intrinsic to the ChatTestMut pipeline. The next sections explore the metrics collected in this study to answer the research questions.

### 4.1 RQ1: How can we compare large- and small-scale models in terms of the number of generated tests?

To answer this question, we consider the number of methods tested and the number of tests per method.

*4.1.1 Number of Methods Tested.* This metric represents the number of methods in the system under test (SUT) for which test code could be generated. For each project, ChatTester first parses all methods and filters them using simple heuristics (e.g., methods must be public, nontrivial, nonvoid, and not from test files). For example, if a project contains 120 methods, ChatTester may consider only 65 of them as "testable." These are then sent to the LLMs to attempt test generation. Table 2 summarizes the tested projects, including their domain, total number of methods, and how many were considered testable by ChatTester (column "Sent Methods"). This restricted the capacity of LLMs as seen in Figure 4, the proportion of tests generated by each LLM and the ones generated by EvoSuit, and the upper bound of LLMs.

EvoSuite outperformed all LLMs in terms of methods tested because it attempts to test all methods, without filtering. Among the large-scale LLMs (the first six in Fig. 4), Gemma, xLAM, and DeepSeek generated tests for more than half of the testable methods. Interestingly, DeepSeek (a small-scale model) even outperformed Gemini (a much larger model from Google), though its total remained slightly below the average for large-scale models (247 methods across all projects). Another relevant observation is that xLAM (1B parameters) surpassed four other small models with more parameters, including Llama3B, demonstrating that size alone is not always a good predictor of method coverage in test generation.

*4.1.2 Tests per Method.* The number of tests per method indicates the size of the test suite generated for a SUT method. As mentioned above, in this study, we asked a set of LLMs to generate tests for seven projects (see Table 2). In this section, we present the mean number of tests per method to consider all the target projects in our analysis.

As we can see in the third column of Table 2, some large-scale models presented the best results, with an average of three tests per method. Among small-scale LLMs, *LLaMA 3.2-3B* also stood out, achieving more than three tests per method. In this metric, EvoSuite outperformed seven small-scale LLMs and the large-scale open-codestral, was roughly tied with four small-scale models, and was surpassed by Llama3B and five large-scale models.

However, large-scale LLMs such as *open-codestral* and *codestral* generated an average of 1.32 and 1.9 tests per method, respectively. Llama3B was the only small-scale LLM that generated more than 2 tests per method, generating 3.44 tests per method. Although this small-scale model did not test the greatest number of methods, it generated a higher number of tests per method compared to large-scale models with 8 to 30 billion parameters.

Three small-scale LLMs (xLan, Gemma, and Deepseek) were able to test as many methods as the large-scale LLMs, and Llama3B outperformed them in the number of tests. However, no small-scale model matched the performance of large-scale models on both metrics simultaneously.

### 4.2 RQ2: How can we compare the quality of the tests generated by small-scale LLMs and large-scale LLMs?

To answer this question, we consider the mutation score, cyclomatic complexity, and smells (average number, proportion, and type).

*4.2.1 Mutation Score.* The mutation score shows the percentage of mutants that were detected (or killed in mutation test terminology) by the generated tests. We could not collect this metric for EvoSuite tests because, as mentioned before, our mutation solution was tied to the ChatTesterMut pipeline.

The mutation score tries to approximate how well a test can detect failures.

As we can see in the fourth column of Table 2, the small-scale LLMs achieved an average score of 62.92%, outperforming the large-scale LLMs, which achieved an average mutation score of 52.32%. This highlights how these small-scale models can be promising in some aspects of automated test generation.

Taking into account all LLMs analyzed in this study, the smaller version of Llama, the *Llama1B*, had the best mutation score with 84.69% of the mutants killed. However, this result should be viewed with caution, as its tests only covered 95 methods. Looking at *xLAM*,*gemma*, and *deepseek*, the three small-scale models that covered as many methods as the large-scale ones, *xLAM* stands out as

**Table 3: Evaluated Metrics**

| Model ID | # Methods Tested | Mean #Tests per Method | Mutation Score | Mean Cyclomatic Complexity | Mean #Test Smells per Test | %Smelly Tests | Most Frequent Smell | Mean #Corrections |
|---|---|---|---|---|---|---|---|---|
| codestral | **282** | 1.90 | 41.16% | **1.02** | 1.87 | 54.38% | Mag. Num. | 1.71 |
| gemini-mini | 253 | 2.77 | 57.65% | 1.21 | 0.26 | 8.06% | Lazy | 1.91 |
| gemini | 225 | **3.15** | 51.24% | 1.18 | **0.00** | **0.00%** | - | 1.73 |
| gpt4o | 246 | 2.67 | 57.80% | 1.08 | 2.26 | 57.77% | Lazy | **1.12** |
| open-codestral | 240 | 1.32 | **60.60%** | 1.05 | 2.07 | 75.87% | Mag. Num. | 2.08 |
| Qwen32B | 236 | 2.36 | 45.48% | 1.04 | 2.80 | 71.31% | Lazy | 1.24 |
| Llama3B | 143 | **3.44** | 65.32% | 1.05 | 3.26 | 75.91% | Lazy | 3.70 |
| granite | 121 | 1.24 | 77.21% | 1.02 | 1.29 | 49.67% | Mag. Num. | 3.25 |
| xLAM | 220 | 1.08 | 70.54% | 1.03 | 1.74 | 75.11% | Mag. Num. | 2.63 |
| gemma | 215 | 1.05 | 58.97% | **1.01** | **0.12** | **6.22%** | Mag. Num. | 2.23 |
| deepseek | **236** | 1.12 | 60.67% | 1.06 | 1.61 | 71.02% | Mag. Num. | 2.08 |
| Yi | 129 | 1.12 | 57.04% | 1.03 | 1.47 | 69.66% | Mag. Num. | **1.41** |
| Smol | 107 | 1.70 | 53.17% | **1.01** | 2.44 | 77.20% | Mag. Num. | 5.23 |
| Qwen1.5B | 127 | 1.57 | 73.47% | 1.03 | 2.43 | 73.37% | Mag. Num. | 4.22 |
| Qwen0.5B | 132 | 1.05 | 71.19% | **1.01** | 1.74 | 74.64% | Mag. Num. | 3.41 |
| OpenCoder | 182 | 1.03 | 48.85% | 1.13 | 2.20 | 89.30% | Mag. Num. | 3.63 |
| Llama1B | 95 | 1.66 | **84.69%** | 1.02 | 2.77 | 83.86% | Lazy | 3.62 |
| Falcon | 41 | 1.61 | 33.93% | 1.07 | 1.86 | 80.30% | Mag. Num. | 6.20 |
| EvoSuite | 542 | 1.61 | - | 1.17 | 3.88 | 59.24% | Lazy | - |

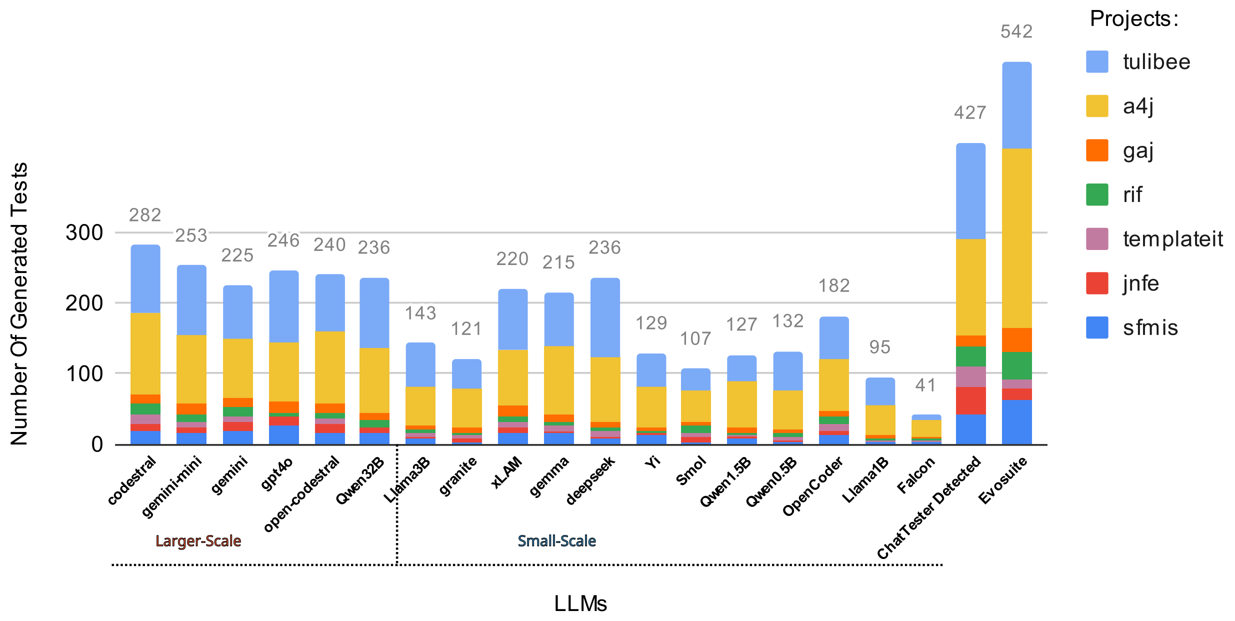

**Figure 4: Tests Generated for Each Project**

the best among them with a mutation score of 70.54% and all three showed scores superior to the average of large-scale LLMs.

Surprisingly, the best large-scale model was an open one, the *open-codestral* performed better than its closed version of the *codestral* (with 60.60% and 41.16% mutation score, respectively).

#### 4.2.2 *Cyclomatic Complexity.*

For the cyclomatic complexity metric, this rule applies: the lower, the better. A lower value of the cyclomatic complexity means that it will be less complex to understand and maintain a given code.

The fifth column of Table 2 shows that the cyclomatic complexity values for the test methods generated by almost all LLMs are lower than the cyclomatic complexity for the methods generated by EvoSuite. Except for *gemini-mini*, which is 4 points higher than EvoSuite. In general, the values of the cyclomatic complexity are very low in all small- and large-scale LLMs.

The only models with a cyclomatic complexity above 1.1 were the large-scale models *gemini*, *gemini-mini*, and *OpenCoder*, making them clear outliers. In contrast, the three best-performing were the small-scale models *gemma*, *Qwen0.5B*, and *Smol*, each achieving a cyclomatic complexity of 1.01.

#### 4.2.3 *Mean Number of Test Smells per Test.*

The number of test smells showed a different result from previous work [36], showing LLMs with fewer smells than EvoSuite, which was also obtained with TsDetect [31]. As previous studies showed [36], the code generated by LLMs presented a high number of test smells. However, we got an exception with Google models *gemini,gemini-mini*, and *gema*. The main *gemini* did not present any test smells. An interesting point is how the Google open model *gemma* outperformed the commercial *gemini-mini* model in this regard, with only 1/4 of its parameters. The LLaMA model from Meta [16], on the other hand, generated the highest number of test smells.

#### 4.2.4 *Proportion of Test Smells.*

Considering the proportion of test smells presented in Fig. 5, Google large-scale LLMs (gemini and gemini-mini) and its small-scale LLM (gemma) obtained by far the best results. All of them have less than 10% of smelly tests. The small-scale IBM model *granite* also performs slightly better than most large-scale LLMs (although it only uses 1 billion parameters).

Taking into account the comparison with EvoSuite, a previous study [36] showed that the proportion of smelly tests was slightly better in EvoSuite compared to ChatGPT. In our study (using different projects), ChatGPT performed slightly differently (57.8%) than EvoSuite (59.2%). In addition, we could observe which LLMs perform better and worse than EvoSuite in this metric.

#### 4.2.5 *Most Frequent Test Smells.*

The four most common smells are listed from left to right in descending order of frequency, as shown in Fig. 6. They are [30]:

- **Magic Number Tests:** instead of a variable with a name that explains the meaning, a constant is used;
- **Lazy Test:** many tests calling the same SUT method
- **Eager Test:** one test call multiple SUT methods
- **Exception Handling:** the test has a throw or a catch statement inside it.

The Exception Handling explains why the EvoSuite had so many Smells. As EvoSuit is targeted at multiple versions of JUnity, it should not rely on the *assertThrows* of JUnity5. However, even if we exclude this type of smell, EvoSuite tests still have more smells per test than the tests generated by LLM.

The most frequent smell among small-scale LLMs was the Magic Number. In large-scale LLMs, the *lazy test* occurred more. The Lazy Test is also the most common smell in EvoSuite.

In the study by Siddiq et al. [36], LLM-generated tests exhibited more test smells than those of EvoSuite, which contrasts with our findings in Table 3, where EvoSuite produced tests with a higher number of smells. Furthermore, Siddiq et al. [36] identified *empty test*, *redundant print*, and *redundant assertion* as the most frequent smells in the LLM-generated tests, while in our evaluation, these were mostly absent from the LLM-generated tests.

> The quality of the tests generated by some small-scale LLMs is comparable to that of large-scale ones. Small-scale LLMs perform well in the mutation score, and some of them perform well in the presence of test smells, notably the *gemma*. However, the cyclomatic complexity shows how small-scale LLMs generate simpler tests compared to the ones generated by large-scale LLMs. Although simpler, their tests are efficient, as demonstrated for mutation scores.

## 4.3 RQ3: How can we compare small- and large-scale LLMs in terms of the number of iterations needed to generate tests?

To answer this question, we consider the number of interactions needed to generate a compilable test.

#### 4.3.1 *Correction Iterations.*

A key limitation of small-scale models is the need for more correction iterations to produce compilable tests. On average, 3.47 correction iterations were needed to generate tests compared to 1.63 iterations when using large-scale models, as can be seen in Fig. 7.

The model that requires fewer corrections is the GPT, as it is the model originally used in the creation of the tool used to benchmark, the ChatTester. The *open-codestral* from Mistral-AI needed an average of 2 corrections, as it was the smallest large-scale model with only seven billion parameters. The Yi-Coder was the small-scale model with the best correction score, which needed only 1.4 corrections to create a compilable test, and the only one with a performance comparable to the large-scale models.

> The tests created by small-scale LLMs require, on average, a higher number of correction iterations. However, two models, *gemma* and *Yi*, achieved a comparable number of corrections, necessitating two or fewer correction iterations.

## 5 Discussion

In this benchmark study of large- and small-scale LLMs, we found that small-scale models outperformed the larger ones in mutation detection and cyclomatic complexity, with Llama3B also generating more tests per method. Another observation was how one model can differ from another, not only because of its number of parameters but also because of how they were trained and implemented. The Google Gemini and Gemma models generated codes with very few smells, which can be explained by their internal directive for code standards [15] and the large Java codebase of Android. The small-scale LLMs showed a frequent number of the *Magic Number* smell, and this smell could be avoided if the prompt used to make the tests explicitly called the model to create a variable instead of using constants. This emphasizes the importance of the prompt in determining the quality of the generated tests. In our analysis, the

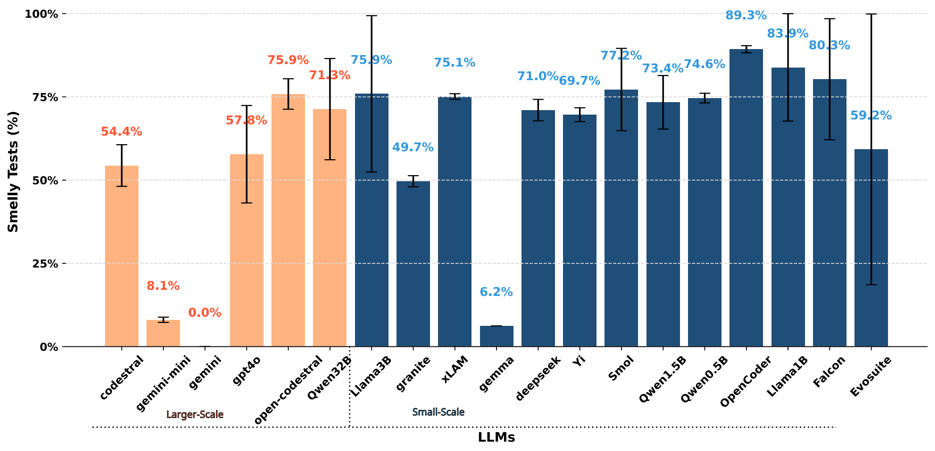

**Figure 5: Proportion of Tests with Smell**

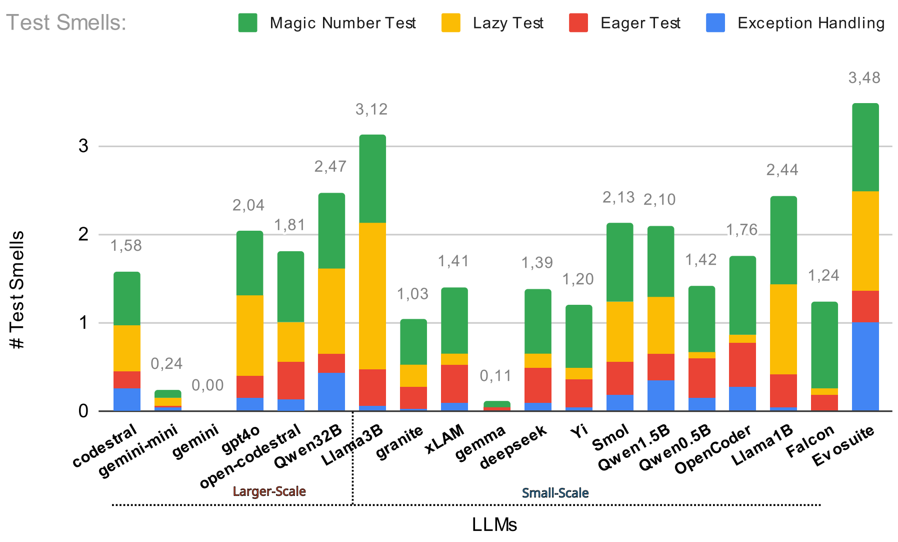

**Figure 6: The 4 Most Common smells**

best small-scale models were *gemma*, *xLAM*, and *deepseek*, although some performed better in some metrics; these 3 could cover as many methods as large-scale models, while maintaining an above-average mutation score and a low cyclomatic complexity.

It is worth mentioning that although the small-scale LLMs needed more iterations for generating compilable test code, such iterations can be less expensive (e.g., in terms of energy consumption) than those of large-scale LLMs, since small-scale models could run on modest hardware locally. On the other hand, the waiting time can become a burden if it requires several iterations to generate a test. Most of the small-scale models in this study needed around two correction iterations.

Although large-scale LLMs trained for coding did not perform much better than generalist models, regarding small-scale models, code-dedicated models outperformed generalist models for test generation, 2 of the 3 best small-scale models were created for coding (Table 1).

Previous studies mentioned that small-scale models could perform equally or better in summarization [14], text classification [6], sentiment analysis [12], and creative writing [24]. The present study suggests that large-scale and small-scale models can achieve similar results regarding test generation, as the latter have already outperformed the former in terms of mutation score.

Given that small-scale models received similar to slightly better results in the evaluated metrics, they emerge as a viable alternative for users with privacy concerns. They are also a more efficient alternative, as their smaller size implies a significantly lower computational cost [33].

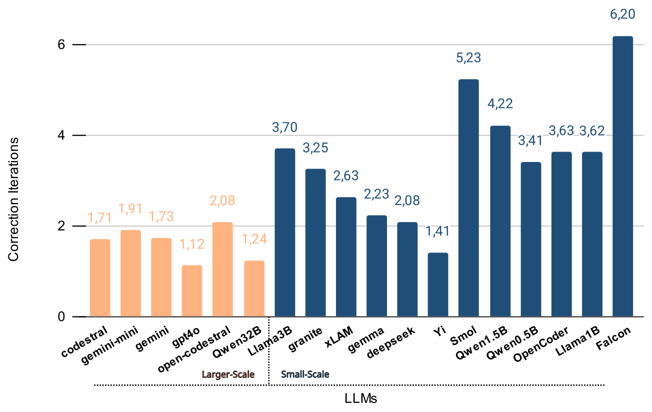

**Figure 7: Correction Iterations**

## 6 Threats to Validity

In this section, we discuss the threats to the validity and how we mitigate them in this work.

### 6.1 Internal Validity

A potential internal threat to validity in our study is selection bias in the choice of models for evaluation. We aim to consider all possible small-scale models in the HuggingFace [11] model repository. However, some open models might not be listed there, or they might use a different naming convention that caused them to be excluded during selection. Another potential threat is the mutation technique implemented in this work. To avoid using potentially broken code, each mutation was first submitted to the same compilation check as the test cases. In addition, all mutations were generated using deterministic code. However, our specific mutation strategy, or the chosen mutations themselves, might have favored certain models over others or failed to reflect certain use cases that some models would handle better.

### 6.2 External Validity

The SF110 benchmark comprises a diverse set of Java projects, but our study uses only seven projects. This could make our selection inside a subset where some LLMs perform better than others. We also verified all used LLMs in the first class and the method of the first project to check if they are capable of generating tests, which could exclude an LLM capable of generating tests from all other cases except this. One benchmark with all 110 projects would be desirable, but it would consume much time and test the limits of the free LLMs API. Hence, as in similar studies [43], our results cannot be generalized to all types of projects, languages, sizes, and domains.

### 6.3 Conclusion Validity

The set of metrics collected in this study may also introduce an additional source of bias. Using the number of iterations without considering the time required to generate the tests can lead to the false impression that one LLM is more efficient than another. A more accurate evaluation would require measuring hardware usage. Ideally, large- and small-scale models should be compared on identical infrastructure. However, in this study, all large-scale models were executed on proprietary cloud platforms.

### 6.4 Construct Validity

Our benchmarking relies on the tools: Lyzard, LLM Python library, TsDetector, ChatTester, and a custom mutation testing implementation. Before using them in the study, we manually validated their output on representative examples to check for consistency and correctness. For example, we excluded TsDetector results for ignore test smell due to inconsistent detection behavior. Moreover, the heuristics of ChatTesterMut are the same as those of ChatTester and could select favorable methods for one type of LLM, as ChatTester was developed to work with the GPT3.5 model. To mitigate this issue, our analyses also focused on the quality of those tests, checking the mutation score and the presence of test smell. Our work did not detect a particular bias from the ChatTester towards the OpenAI model.

## 7 Conclusion

This study compared small-scale LLMs with large-scale ones to test generation. Our analysis shows that some small-scale LLMs, such as gemma2, xLAM1b, and deepseek-coder, achieved results comparable to large-scale models in terms of test generation. Although small-scale LLMs generally produced tests with more smells, all LLMs outperformed EvoSuite in code quality. Surprisingly, the tests generated by the selected small-scale LLMs were more effective in detecting mutations than those of the large-scale models. This study can be used as a reference point for LLM-generated tests benchmark. All codes used are made available to facilitate their replication. We also make our data available for further analysis.

Future work could investigate the design of prompts specifically aimed at reducing test smells. In addition, explore the combination of two or more small-scale LLMs to take advantage of their complementary strengths, potentially outperforming a single large-scale LLM in multiple quality dimensions. Last but not least, this study suggests that the benefits of generative AI can be democratized through small-scale open-source models. This enables local execution, preserves user confidentiality, and is particularly relevant for settings such as small software companies and small independent game development teams. In addition, it can reduce energy costs, making generative AI more accessible and sustainable.

## ARTIFACT AVAILABILITY

All information needed to rerun this experiment can be found at: https://zenodo.org/records/17046209. If you only want to check the benchmark data, look at the finalcompleddata.xlsx file in this repository.

## ACKNOWLEDGMENTS

We thank NPAD/UFRN for their computing resources. This work is partially funded by INES (https://www.ines.org.br). This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil* (CAPES) – Finance Code 001.

## REFERENCES

[1] Mistral AI. 2024. Open Codestral Mamba (open-codestral-mamba). https://huggingface.co/mistralai/Mamba-Codestral-7B-v0.1. https://huggingface.co/mistralai/Mamba-Codestral-7B-v0.1 Open-source code model based on the Mamba2 architecture, released in July 2024.

[2] Mistral AI. 2025. Codestral 25.01. https://mistral.ai/news/codestral-2501. https://mistral.ai/news/codestral-2501 Code generation language model with optimized architecture and improved tokenizer.

[3] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology* 176 (2024), 107565.

[4] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. 2025. SmolLM2: When Smol Goes Big – Data-Centric Training of a Small Language Model. arXiv:2502.02737 [cs.CL] https://arxiv.org/abs/2502.02737

[5] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).

[6] Martin Juan José Bucher and Marco Martini. 2024. Fine-Tuned'Small'LLMs (Still) Significantly Outperform Zero-Shot Generative AI Models in Text Classification. *arXiv preprint arXiv:2406.08660* (2024).

[7] Henry Coles, Matjaz Jurcenoks, Peter Reilly, and Emma Armstrong. 2016. PIT Mutation Testing Tool. https://pitest.org Available at: https://pitest.org.

[8] Google DeepMind. 2024. Gemini 1.5 Flash. https://blog.google/technology/ai/google-gemini-update-flash-ai-assistant-io-2024/. https://blog.google/technology/ai/google-gemini-update-flash-ai-assistant-io-2024/ Lightweight multimodal language model optimized for speed and efficiency, featuring a 1 million token context window.

[9] Google DeepMind. 2024. Gemini 1.5 Flash-8B. https://ai.google.dev/gemini-api/docs/models/gemini. https://ai.google.dev/gemini-api/docs/models/gemini Multimodal language model optimized for high-volume, lower-intelligence tasks; supports audio, image, video, and text inputs with a 1 million token context window.

[10] Brad Everman, Trevor Villwock, Dayuan Chen, Noe Soto, Oliver Zhang, and Ziliang Zong. 2023. Evaluating the carbon impact of large language models at the inference stage. In *2023 IEEE international performance, computing, and communications conference (IPCCC)*. IEEE, 150–157.

[11] Hugging Face. 2025. Hugging Face. https://huggingface.co

[12] Sorouralsadat Fatemi and Yuheng Hu. 2023. A comparative analysis of fine-tuned LLMs and few-shot learning of LLMs for financial sentiment analysis. *arXiv preprint arXiv:2312.08725* (2023).

[13] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[14] Xue-Yong Fu, Md Tahmid Rahman Laskar, Elena Khasanova, Cheng Chen, and Shashi Bhushan TN. 2024. Tiny titans: Can smaller large language models punch above their weight in the real world for meeting summarization? *arXiv preprint arXiv:2402.00841* (2024).

[15] Google. 2023. Google Java Style Guide. https://google.github.io/styleguide/javaguide.html Accessed: 2025-04-07.

[16] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[17] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv* abs/2401.14196 (2024). https://arxiv.org/abs/2401.14196

[18] Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. 2021. Natural language-guided programming. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 39–55.

[19] Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2024. OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models. https://arxiv.org/pdf/2411.04905

[20] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[21] Dacheng Li, Rulin Shao, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. How long can context length of open-source llms truly promise?. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

[22] Xiang-Jun Liu, Ping Yu, and Xiao-Xing Ma. 2024. An Empirical Study on Automated Test Generation Tools for Java: Effectiveness and Challenges. *Journal of Computer Science and Technology* 39, 3 (2024), 715–736.

[23] Nils Lukas, Ahmed Salem, Robert Sim, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2023. Analyzing leakage of personally identifiable information in language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 346–363.

[24] Guillermo Marco, Luz Rello, and Julio Gonzalo. 2024. Small language models can outperform humans in short creative writing: A study comparing slms with humans and llms. *arXiv preprint arXiv:2409.11547* (2024).

[25] Matt Cone. 2020. Markdown Guide. https://www.markdownguide.org.

[26] Aishwarya Narasimhan, Krishna Prasad Agara Venkatesha Rao, et al. 2021. Cgems: A metric model for automatic code generation using gpt-3. *arXiv preprint arXiv:2108.10168* (2021).

[27] Tung Nguyen. 2024. Lizard - A Code Complexity Analyzer Without Pylint. https://pypi.org/project/lizard/ Accessed: 2024-04-06.

[28] OpenAI. 2024. GPT-4o Mini. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ Lightweight multimodal language model optimized for cost-efficiency and performance, supporting text and vision inputs with a 128K token context window.

[29] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[30] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 193–202.

[31] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. TsDetect: An Open Source Test Smells Detection Tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1650–1654. doi:10.1145/3368089.3417921

[32] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. 2022. Learning To Retrieve Prompts for In-Context Learning. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz (Eds.). Association for Computational Linguistics, Seattle, United States, 2655–2671. doi:10.18653/v1/2022.naacl-main.191

[33] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[34] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 30–34.

[35] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[36] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. 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*. 313–322.

[37] Falcon-LLM Team. 2024. The Falcon 3 Family of Open Models. https://huggingface.co/blog/falcon3

[38] Gemma Team. 2024. Gemma. (2024). doi:10.34740/KAGGLE/M/3301

[39] IBM Granite Team. 2024. Granite 3.1: Powerful Performance, Longer Context, and More. *IBM Research Journal* 47, 4 (2024), 22–29. https://www.ibm.com/new/announcements/ibm-granite-3-1-powerful-performance-long-context-and-more

[40] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[41] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[42] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, et al. 2024. Yi: Open foundation models by 01. ai. *arXiv preprint arXiv:2403.04652* (2024).

[43] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[44] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging.* Morgan Kaufmann.

[45] Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, et al. 2024. xlam: A family of large action models to empower ai agent systems. *arXiv preprint arXiv:2409.03215* (2024).