Evaluating the Effectiveness and Cost-Efficiency of Large Language Models in Automated Unit Test Generation

Werney Ayala Luz Lira
Federal Institute of Piauí
Pedro II, Brazil
werney@ifpi.edu.br

Pedro de Alcântara dos Santos Neto
Federal University of Piauí
Teresina, Brazil
pasn@ufpi.edu.br

Guilherme Amaral Avelino
Federal University of Piauí
Teresina, Brazil
gaa@ufpi.edu.br

Luiz Fernando Mendes Osório
Federal University of Piauí
Teresina, Brazil
fernando.mendes@ufpi.edu.br

ABSTRACT

Software quality plays a crucial role in the development process, and one of the main strategies to ensure it is through software testing. However, testing can be a costly and time-consuming task, leading researchers to explore ways to automate it. This study analyzes the ability of large language models (LLMs) to generate unit tests automatically, evaluating their performance based on four key criteria: line coverage, branch coverage, hit rate, and cost. To achieve this, unit tests were generated using three different input formats: (i) source code only, (ii) source code with a docstring, and (iii) a detailed prompt with step-by-step instructions. The results show that all models produce tests with high line and branch coverage. However, hit rates vary depending on the input format. The lowest hit rate was 72.61%, obtained by ChatGPT-4o Mini when provided with only the source code. The highest hit rate was 90.69%, achieved by Gemini 1.5 Pro when given source code with a docstring, a performance close to the 96.81% hit rate of the dataset's reference tests. The findings indicate that more affordable models, such as Gemini 1.5 Flash and GPT-4o Mini, can achieve competitive results when optimized prompts are used, making them cost-effective alternatives for unit test generation.

KEYWORDS

Software Testing, Automatic Software Testing, Large Language Models, LLM, Chatgpt, Gemini, Artificial Intelligence

1 Introduction

Software quality is one of the most crucial factors in ensuring that a computing solution is secure, reliable, easy to maintain and use, has good performance, among other aspects [38]. One key technique for maintaining software quality is testing [18]. There are various testing approaches, ranging from unit testing, which examines individual modules in isolation [20], to integration testing, which assesses the interaction between interconnected modules [27], and regression testing, which verifies that changes in the code have not generated errors in functionalities that previously worked [14].

Unit testing [4][44] aims to verify whether a unit is functioning correctly. Typically, unit tests are manually created by the development team, making it a time-consuming and resource-intensive task [11]. Even though it is a costly activity, unit testing contributes to the early identification of errors, facilitates code maintenance, and increases the reliability of the developed features. As a result, development teams can reduce rework, boost productivity, and ensure higher-quality deliveries [24].

One way to reduce the costs and time involved in this activity is through the automated unit test generation. [52]. Over the recent years, numerous researchers have dedicated efforts to developing various approaches for automating testing activities [3]. Among the techniques employed are random test generation [39] [35], fuzzy systems [45], evolutionary algorithms [9], model-based test generation [46], and search-based test generation [16] [17], among others.

However, many of these techniques face challenges related to the readability of generated tests [22] or the difficulty of generating tests across multiple programming languages [36]. Manual tests, on the other hand, do not face these issues since they are created by developers specialized in this activity. Thus, it can be observed that both manual and automatic generation of unit tests have their advantages and disadvantages [43].

With advances in artificial intelligence and the growing popularity of tools like ChatGPT, some researchers have begun exploring the use of Large Language Models (LLMs) [47] for software test generation [2] [41]. These models have shown significant promise in similar tasks, with one notable example being GitHub Copilot[1], a tool that leverages LLMs to assist developers in coding tasks [26] [33] [31].

The application of LLMs in automatic unit test generation could help overcome key challenges in this field, such as code readability [51] and the ability to generate tests in multiple programming languages [36]. In addition to the advantages already offered by conventional techniques, such as generating tests more quickly and ensuring good code coverage, LLMs introduce new possibilities for improving software testing automation [32]. However, there are still gaps in the literature regarding a detailed understanding of the behavior of these models, especially with respect to the impact of different prompt strategies on the quality of the generated tests and the associated costs.

The main objective of this study is to evaluate the effectiveness and cost-benefit ratio of different large language models (LLMs) in the automatic generation of unit tests from source code. While previous studies have explored the ability of LLMs to generate tests automatically, little attention has been given to how different types

[1]https://github.com/features/copilot

of prompts influence the quality of the generated tests and how prompt size affects associated costs.

This work stands out by adopting an evaluation approach that, in addition to considering traditional metrics such as line and branch coverage, incorporates hit rate and a detailed cost analysis, in order to provide a more comprehensive view of the effectiveness and feasibility of using LLMs for automatic unit test generation.

Furthermore, this study compares three levels of prompt complexity (a simple prompt, a prompt with a docstring, and a detailed prompt) with the goal of analyzing how prompt design affects test quality and how prompt length impacts cost. This allows for the identification of combinations between model and input that maximize test quality at the lowest possible cost.

The remainder of this paper is organized as follows. Section 2 presents related work on the topic, Section 3 presents the concepts necessary for a proper understanding of the study, Section 4 describes the methodology applied in this study, Section 5 discusses the results obtained, Section 6 discusses limitations and threats to validity, and Section 7 presents the conclusion and directions for future work.

2 Related Works

This section presents studies and research related to the topic addressed in this work. The following analysis was conducted to identify the contributions of each study, the methodologies used, and the results obtained in previous research. Additionally, key gaps and challenges in the field were discussed, highlighting the relevance and necessity of the current study. A brief survey listing the most commonly used techniques is also included.

The study by [42] provides an empirical evaluation of the effectiveness of automatically generating unit tests using LLMs. The author emphasizes that no additional training or manual effort was involved. In this study, the models were given a function along with usage examples to generate tests. An extra step was also included to check for errors in the generated tests and prompt the model to correct any issues. The approach was tested using the GPT-3.5 Turbo model, achieving 70.2% line coverage and 52.8% branch coverage. These results outperformed the feedback-directed test generation technique, which achieved only 51.3% line coverage and 25.6% branch coverage.

The study by [12] introduces MuTAP as a strategy to enhance the effectiveness of test cases generated by LLMs. To achieve this, prompts are refined using surviving mutants, as these mutants highlight the limitations of test cases in detecting changes in a program. The results showed that the proposed method can detect up to 28% more faulty human-written code snippets. Among these, 17% remained undetected by both the state-of-the-art fully automated test generation tool (Pynguin) and the zero-shot/few-shot learning approaches in LLMs.

In [51], an empirical study was conducted to assess ChatGPT's ability to generate unit tests. The study included a quantitative analysis and a user study to evaluate the quality of the generated tests in terms of correctness, sufficiency, readability, and usability. The findings revealed that ChatGPT-generated tests still suffer from correctness issues, ranging from compilation errors to runtime failures. On the other hand, in terms of coverage and readability, they are very similar to manually written tests. Finally, the authors propose ChatTester, an approach that leverages ChatGPT to generate tests and iteratively refine them. The results showed that the proposed approach produces tests that are 34.3% more likely to compile and 18.7% more likely to have correct assertions compared to standard ChatGPT-generated tests.

In [36], a thorough analysis was conducted on how LLMs can automatically generate unit tests for multiple programming languages while maintaining good readability. The study describes a pipeline that incorporates static analysis to guide the models in generating high-coverage test cases. The results demonstrated that LLM-based test generation can be competitive with, and even surpass, state-of-the-art test generation techniques in terms of achieved coverage, while also producing significantly more natural test cases.

This study stands out by exploring strategies to make the use of these tools more efficient and cost-effective. The related works cited here generally focus on metrics such as coverage and accuracy. However, few assess the costs associated with using these models or explore ways to optimize prompts to improve cost-effectiveness. The proposed study differs by considering not only the effectiveness of the models but also their feasibility, identifying approaches that enhance their practical use in the software industry.

Another distinguishing factor is the experimental approach applied in this work, which will be presented in the next section. While many studies use only one type of input for the models, this work examines three different input formats: raw source code, source code with a docstring, and a detailed prompt. This allows for a more comprehensive analysis of how different types of instructions impact model performance, contributing to a better understanding of how to optimize their use.

3 Background

This section aims to present the fundamental concepts that support this study, providing the theoretical foundation needed to understand the problem under investigation. It covers topics such as unit testing, its benefits and challenges, as well as techniques and tools for its automatic generation. The section also discusses the concept of Large Language Models (LLMs), highlighting how they work, how they are built, and their potential applications in the context of software engineering. Finally, it introduces the HumanEval dataset, which serves as a benchmark in this study for assessing the ability of LLMs to automatically generate unit tests.

3.1 Unit Tests

Unit tests are one of the most important practices in the software quality assurance process [24]. Their main goal is to verify whether individual units of code behave as expected. To achieve this, unit tests focus on small parts of the system, such as functions or methods, in isolation. This allows defects to be detected early, so they can be addressed during the initial stages of development, reducing both costs and risks in later phases of the project [1]. A robust unit test suite not only facilitates the identification and correction of bugs but also supports the introduction of new features and helps detect regressions [7].

Despite their importance, writing unit tests is not a trivial task. Developers need a solid understanding of the code's structure and logic in order to identify input values that will exercise a wide range of execution paths. This ensures that the generated tests achieve comprehensive coverage of the code under test [8] [30].

Even when developers are familiar with the code, it is not always easy to determine which cases should be tested to cover both expected behaviors and edge cases. For this reason, automatic unit test generation has become an increasingly relevant research area, especially with the rise of artificial intelligence techniques and, more recently, the use of large language models (LLMs) [52].

It is worth noting that automatic unit test generation techniques and tools should be understood as auxiliary resources in the software quality assurance process. While they provide significant advantages, especially in terms of saving time, they do not replace the developer's role in identifying implicit requirements and analyzing non-trivial behaviors. Automation can accelerate the process and suggest initial test cases, but developer validation and analysis remain essential to ensure the effectiveness of the generated tests.

3.2 Large Language Models

Large Language Models (LLMs) are built using deep learning techniques [21] trained on massive volumes of data. The purpose of this training is to extract complex patterns from natural language, which enables these models to perform a wide range of natural language processing tasks, such as text translation, summarization, text generation, question answering, and even code generation.

Most of these models are based on the transformer architecture [47], which is characterized by its ability to parallelize the training process. This parallelization helps reduce the time required for training, especially given the large amount of data involved. Typically, the training data consists of text gathered from the internet, books, articles, and other sources.

In recent years, LLMs have been increasingly applied to technical domains such as software development, where they can assist with tasks like code generation, code explanation, bug fixing, and automatic test generation. However, their performance strongly depends on how the instructions are formulated (prompt engineering) and the quality of the data used for training. For this reason, their use requires careful handling and developer oversight.

This study selected four LLMs for evaluation: two models from Google's Gemini family and two from OpenAI's GPT family. Specifically, the models used were Gemini 1.5-pro, Gemini 1.5-flash, ChatGPT 4o, and ChatGPT 4o-mini. These models differ significantly in terms of cost and performance, making them suitable for a comparative analysis that considers both effectiveness and the economic feasibility of automatic test generation.

Another important factor is that all selected models are publicly available, which ensures that the experiments conducted in this study can be easily replicated by other researchers. The availability of lightweight versions, such as the "mini" and "flash" variants, was also a criterion for selection. This allowed us to investigate whether more affordable and less resource-intensive models can compete with more powerful alternatives when prompt engineering strategies are applied.

Thus, the choice of these models enables a balanced analysis across robustness, accuracy, and computational cost, supporting the development of more rational strategies for applying LLMs in software engineering with attention to both effectiveness and cost-efficiency.

3.3 Dataset

HumanEval [10] is a dataset designed to evaluate the capabilities of language models in programming tasks, particularly in automatic code generation. It consists of 164 independent problems written in Python. These problems were carefully selected to assess basic programming skills, such as list and string manipulation, recursion, and the use of conditional and loop structures.

Each item in the dataset includes three main components: a function signature accompanied by a docstring that describes the expected behavior in natural language, along with examples of inputs and their corresponding expected outputs; a function that implements the described behavior; and a set of test cases used to verify the correctness of the implementation.

The fact that each problem can be executed independently allows for the analysis of each case in isolation. This enables the precise measurement of line and branch coverage, as well as hit rate, ensuring fair comparisons between models and prompting strategies.

3.4 Dataset Complexity

To better understand the complexity and size of the problems contained in the dataset, an analysis was conducted using two metrics: cyclomatic complexity [13] to individually measure the complexity of each problem, and lines of code (LOC) [6] to individually measure the size of each problem.

These two metrics were chosen because, when combined, they provide a reliable means of evaluating code [40] [23], particularly in small code snippets consisting of a single method, which is the case for the problems in this dataset. Figure 1 presents a visualization of the distribution of problems based on their cyclomatic complexity and LOC.

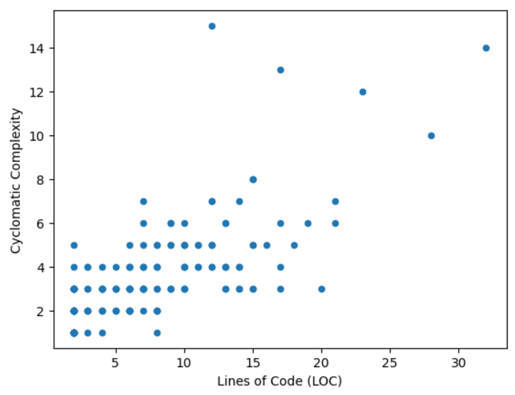

Figure 1: Complexity of dataset problems

The measured cyclomatic complexity values range from 1 to 15, while the LOC values range from 2 to 32. From the analysis in Figure 1, it can be observed that although some problems reach a complexity of 15, the majority fall below 8, with an average cyclomatic complexity of 3.61. For LOC, most problems contain fewer than 20 lines, with an average LOC of 7.78, despite a maximum value of 32.

Therefore, it is evident that the overall complexity of the dataset is relatively low. Although a few problems exhibit higher complexity or size, they represent a small fraction of the 164 problems in the dataset. Specifically, only 5 problems have a cyclomatic complexity greater than 8, and only 5 problems exceed 20 lines of code.

4 Methodology

This study was conducted with the objective of evaluating the effectiveness and cost-efficiency of different large language models (LLMs) in the automatic generation of unit tests. To achieve this, an empirical experiment was structured based on three variations of input prompts and four distinct LLMs, as described below.

Four LLMs were selected for evaluation: Gemini 1.5-pro, Gemini 1.5-flash, ChatGPT 4o, and ChatGPT 4o-mini. These models were chosen due to their current prominence in the landscape of LLM-based tools, combining performance, popularity, and accessibility. Earlier models, such as ChatGPT 3.5-turbo, were also considered. However, since its results diverged significantly from the other models, it was excluded from the comparative analysis.

The HumanEval dataset [10] was used as the source of code for test generation. This dataset is widely adopted for evaluating LLMs in code generation tasks. It consists of 164 carefully designed programming problems. Each problem includes an identifier, a docstring describing the expected behavior of a function along with usage examples, the function implementation, and a set of unit tests to verify its correct behavior.

The evaluation involved submitting all 164 problems from the dataset to the selected models so that unit tests could be generated for each of them. To facilitate this, two prompt templates were created: a simple prompt and a detailed prompt.

The simple prompt provides a direct instruction to the model, as illustrated in the following example: "You are a helpful coding assistant that responds only with Python code. Generate a test class that tests the following code." The detailed prompt, in turn, was constructed using prompt engineering techniques [50], including role prompting, goal specification, task decomposition, and few-shot prompting, as described below.

The first step in constructing the detailed prompt was defining a persona (role) to instruct the model to act as an agent specialized in a specific task—in this case, unit test generation. This was done to provide context for the model and constrain its response in alignment with the objective of the study. The following shows how this technique was applied in the detailed prompt.

Role Prompting

- You are a code generation assistant specializing in the development of unit tests, responding only with Python code.

Next, the task objectives were defined, providing the model with a clear explanation of what it should do, what resources it should use, and what is expected as the final output. This included constraints, the desired output format, and quality criteria. The application of this technique in the detailed prompt is shown below.

Goal Specification

- We are creating unit tests to verify and identify errors or failures in source code.
- To accomplish this task, the source code to be tested is provided as input.
- At the end, the expected response is a class containing as many test cases as necessary to ensure that the code has no errors or failures.

The task was also decomposed with the goal of breaking down complex activities into simpler, more manageable steps. This allows the model to follow a well-defined sequence of actions and effectively reach the final objective, generating a unit test. Below are the decomposed and logically ordered tasks used in the prompt.

Task Decomposition

(1) Carefully analyze the provided code.
(2) Identify the possible input types and values.
(3) Identify the possible output types and values.
(4) Understand the core logic of the code.
(5) Based on this analysis, generate a class containing as many test cases as necessary to ensure that the code has no errors or failures.
(6) Ensure that the generated test achieves maximum code coverage.
(7) Pay close attention to the example response—it should contain only the test class written in the same programming language and nothing else.

Finally, the model was presented with an example of input (source code) and an example of the expected output (unit test). The purpose of this step was to help the model understand the expected input-output pattern so that it could generate responses more aligned with the provided structure. Below is an illustration of how this technique was applied in the detailed prompt.

Few-shot Prompting

~~~
### Example Code:

```python
from typing import List

"""
Return list of all prefixes from shortest to longest of the
↪  input string
>>> all_prefixes('abc')
['a', 'ab', 'abc']
"""
```

```python
def all_prefixes(string: str) -> List[str]:
    result = []
    for i in range(len(string)):
        result.append(string[:i+1])
    return result
```

### Example Response:

```python
import unittest

class AllPrefixesTest(unittest.TestCase):
    def test_empty_string(self):
~~~

```
        self.assertEqual(all_prefixes(""), [])

    def test_single_character(self):
        self.assertEqual(all_prefixes("a"), ["a"])

    def test_multiple_characters(self):
        self.assertEqual(all_prefixes("abc"), ["a", "ab",
        ↪  "abc"])

    def test_repeated_characters(self):
        self.assertEqual(all_prefixes("aa"), ["a", "aa"])

    def test_numeric_string(self):
        self.assertEqual(all_prefixes("123"), ["1", "12",
        ↪  "123"])

    def test_special_characters(self):
        self.assertEqual(all_prefixes("!@#"), ["!", "!@",
        ↪  "!@#"])
```
```

The simple and detailed prompts were combined with different levels of contextual information to evaluate how the degree of guidance affects the quality and cost of the generated tests. The following are the three input configurations used in the experiments:

- Input 1 – Simple prompt + source code
- Input 2 – Simple prompt + source code + docstring
- Input 3 – Detailed prompt + source code + docstring

Each of the three input types was designed to address one of the research questions defined below:

- RQ1 - Are the models capable of generating unit tests using only the source code of the program to be tested?
  This question aims to assess the ability of LLM models to automatically generate unit tests with good code coverage and hit rates when provided with the simple prompt presented in the previous section, along with the source code for which the test should be generated.
- RQ2 - Are the generated tests improved when a docstring is added to the source code?
  This question aims to determine whether the quality of the generated tests improves when, in addition to the simple prompt and source code, the models are also provided with a docstring that describes the problem to be solved and includes some input and expected output examples.
- RQ3 - Do the results improve when using a prompt that explicitly details what should be done?
  This question aims to verify whether there is a significant improvement in the quality of the generated tests when the models are provided with a detailed prompt, which outlines each step to be followed, along with input (source code and docstring) and expected output (unit tests) examples.
- RQ4 - Which model offers the best cost-benefit ratio for automated unit test generation?
  This question analyzes the costs associated with generating unit tests for all problems in the selected dataset across each of the scenarios described above.

To address the research questions RQ1, RQ2, and RQ3, inputs 1, 2, and 3 were used, respectively. For RQ4, which involves a broader cost analysis, including which type of input provides the best cost-benefit ratio, all previous configurations were considered.

To analyze the results obtained for each research question, the following metrics were employed: line coverage, branch coverage, hit rate, and associated costs. The coverage metrics were calculated using the Python Coverage library[2], which is widely used to measure the effectiveness of unit tests. Line coverage measures the proportion of code lines executed during testing, while branch coverage considers the different execution paths followed by the tests. Ideally, both values should be close to 100%.

The hit rate was calculated as the ratio between the number of tests that passed and the total number of tests generated. In other words, for each problem, all generated tests were executed and those that passed successfully were counted.

It is well known that a test may fail for two main reasons: either due to a flaw in the test itself or because of an error in the code being tested. However, this study did not investigate the cause of the failures. The analysis focused solely on evaluating the model's ability to generate tests aligned with the expected behavior of the source code. Therefore, test failures were interpreted as limitations in the test generation process.

Finally, the associated costs were estimated using pricing information published by the companies responsible for the models, based on the average number of tokens used in the input and generated in the output. The number of tokens for both input and output was retrieved from the model's own response logs.

4.1 Experimental Design

The following section presents the experimental design. This experiment aims to evaluate the effectiveness and cost-efficiency of large language models (LLMs) in the automatic generation of unit tests. To this end, a 4×3 factorial experimental design was adopted, in which the evaluated factors are:

- Models (4 levels)
  - Gemini 1.5 Flash
  - Gemini 1.5 Pro
  - ChatGPT 4o Mini
  - ChatGPT 4o
- Inputs (3 levels)
  - Input 1 – Simple prompt + source code
  - Input 2 – Simple prompt + source code + docstring
  - Input 3 – Detailed prompt + source code + docstring

The experiment was structured based on the hypothesis that the level of prompt detail provided to LLMs directly impacts the quality of the generated unit tests and the associated costs. It is expected that more elaborate prompts result in more accurate tests, reflected in higher code coverage and success rates, even if this may lead to increased costs. The hypotheses and variables are defined below.

Hypotheses

- H1 – More detailed prompts produce tests with higher code coverage and higher success rates.
- H2 – The cost of test generation increases proportionally with the prompt size.

Variables

[2]https://pypi.org/project/coverage/
```

- Independent Variables:
  - LLM model
  - Input type
- Dependent Variables:
  - Line coverage
  - Branch coverage
  - Hit Rate
  - Cost (in USD per 1 million tokens)

All 164 problems from the dataset were used in their original form. No modifications or filters were applied. Unit tests were generated automatically for each of the 12 possible combinations (4 models × 3 input types).

## 5 Results

This section presents the results obtained from automated unit test generation using LLM models. The results showed that the selected LLM models are capable of generating tests with good code coverage while achieving high hit rates. Furthermore, this process is carried out quickly and at a relatively low cost, as detailed below.

The following sections present the results obtained for each of the research questions mentioned above.

### 5.1 RQ1 - Are the models capable of generating unit tests using only the source code?

As previously described, the first research question is related to the ability of the selected models to generate unit tests. For this first question, the simple prompt ("You are a helpful coding assistant that responds only with Python code. Generate a test class that tests the following code") was provided along with the source code, similar to the example presented in Figure 2. This approach aimed to assess the models' ability to understand the code and generate unit tests correctly based on a straightforward instruction and the source code associated with the problem's solution.

```python
from typing import List
def has_close_elements(numbers: List[float], threshold:
↪  float) -> bool:
  for idx, elem in enumerate(numbers):
    for idx2, elem2 in enumerate(numbers):
      if idx != idx2:
        distance = abs(elem - elem2)
        if distance < threshold:
          return True
        return False
```

Figure 2: Example source code sent along with the simple prompt

Each of the solutions for the problems in the dataset was provided as input to the five LLM models which are, Gemini 1.5 Flash, Gemini 1.5 Pro, ChatGPT 4o Mini, ChatGPT 4o and ChatGPT 3.5 Turbo. The ChatGPT 3.5 Turbo model is a slightly older model, but due to its relevance for having been used in several works [42] [52] [5] [34] [28] [48] [32] [49] [19], it was included in this study.

After submitting all solution codes for the problems, the values for the time taken by the model to generate the test class were calculated, along with line coverage, branch coverage, hit rate, and cost. These results are displayed in Table 1. The presented values represent the averages obtained across all 164 problems.

Table 1: Results presented by LLM models when generating unit tests using only source code as input

LLM Model	Time	Coverage	Branch Coverage	Hit Rate	Total Price
Gemini 1.5 Flash	2,83	98,48%	97,89%	74,47%	$ 0,0131
Gemini 1.5 Pro	7,23	99,27%	98,95%	81,88%	$ 0,4730
ChatGPT 4o Mini	5,20	98,97%	98,54%	72,61%	$ 0,0284
ChatGPT 4o	6,83	98,29%	98,98%	83,31%	$ 0,7307
ChatGPT 3.5 Turbo	2,98	96,03%	95,10%	52,90%	–
HumanEval	–	99,33%	99,06%	96,81%	–

As shown in Table 1, line coverage values were close to 99%, while branch coverage values were around 98% for all tested models. The time taken by each model to generate a test for a given problem was also analyzed. In this regard, the Gemini 1.5-Flash model stands out for its efficiency, taking an average of 2.8 seconds to complete the task. In contrast, the Gemini 1.5-Pro model takes approximately 7.2 seconds on average, which represents an increase of more than 250% in processing time.

The hit rate metric presented in the Table 1 represents the percentage of tests that passed successfully. It is observed that the fastest model, Gemini 1.5-Flash, achieved a hit rate of 74.47%, while ChatGPT 4o reached 83.31%, representing an improvement of nearly 12% in hit rate.

As discussed earlier, the ChatGPT 3.5-Turbo model was also included in this analysis. However, its results were significantly worse than those of the other models. Although it achieved results close to 96% and 95% for the line coverage and branch coverage metrics respectively and took about 2.98 seconds on average to generate the answers, its hit rate was only 52.9%. This value is approximately 20% lower than that of the second worst performing model. This was the main reason why this model was not included in the further analyses.

Additionally, the results obtained from the tests included in the dataset itself were also considered in the evaluation. These results are presented in the "HumanEval" row of Table 1. It is observed that some models already achieve values close to the HumanEval benchmarks in terms of line and branch coverage, but there is still room for improvement in the hit rate.

The cost in dollars for generating unit tests for all the problems in the dataset was also included. The Gemini models proved to be more cost-effective, with an associated cost of about half that of the GPT models. Section 5.4 provides a more detailed breakdown of the costs involved in generating tests for this dataset. They were included in this section primarily to help the reader better assess the relationship between cost and hit rate.

5.2 RQ2 - Do the results improve when adding the docstring to the source code?

A new stage of automated unit test generation was conducted. This time, the source code of the solution was provided along with a docstring describing the expected behavior of the method as input for the models. The Figure 3 presents an example of the input. The text between lines 3 and 9 represents the docstring that was added.

```
1   from typing import List
2
3   """
4   Check if in given list of numbers, are any two numbers
    ↪  closer to each other than given threshold.
5   >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
6   False
7   >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0],
    ↪  0.3)
8   True
9   """
10  def has_close_elements(numbers: List[float], threshold:
    ↪  float) -> bool:
11    for idx, elem in enumerate(numbers):
12      for idx2, elem2 in enumerate(numbers):
13        if idx != idx2:
14          distance = abs(elem - elem2)
15          if distance < threshold:
16            return True
17          return False
```

Figure 3: Example source code with docstring

As observed above, line 4 contains a brief description of the expected behavior of the following code, while lines 5 to 8 include examples of method calls with their respective input values and expected outputs. In addition to the code described above, the simple prompt ("You are a helpful coding assistant that responds only with Python code. Generate a test class that tests the following code") was provided to guide the models regarding the task they were expected to perform. The results obtained after requesting the models to generate unit tests for all 164 problems are shown below in Table 2.

Table 2: Results presented by LLM models when generating unit tests using the simple prompt with source code and docstring

LLM Model	Time	Coverage	Branch Coverage	Hit Rate	Total Price
Gemini 1.5 Flash	1,95	99,55%	99,31%	89,77%	$ 0,0168
Gemini 1.5 Pro	10,67	99,42%	99,18%	90,69%	$ 0,6220
ChatGPT 4o Mini	5,01	99,03%	98,77%	75,75%	$ 0,0288
ChatGPT 4o	4,96	99,66%	99,45%	87,66%	$ 0,7664
HumanEval	–	99,33%	99,06%	96,81%	–

From Table 2, it is possible to observe that the average time required by Gemini 1.5 Pro to generate a unit test with this type of input increased by approximately 3 seconds, whereas for Gemini 1.5 Flash, the time decreased by about 1 second. The code coverage values also improved, getting even closer to 100%.

There was also a significant improvement in the hit rate, especially for the Gemini models, both 1.5 Pro and 1.5 Flash. The hit rate for the 1.5 Pro model increased from 81.88% to 90.69%, an improvement of nearly 11% (8.8 percentage points). The improvement for the 1.5 Flash model was even greater, with an increase of approximately 15.3 percentage points, rising from 74.47% to 89.77% simply by adding the docstring in the code.

Regarding costs, it was observed that the value remained almost unchanged, with a slight increase for the more expensive models (Gemini 1.5 Pro and ChatGPT 4o). This is due to the fact that the number of tokens added to the input prompts was small, as illustrated in Figure 3.

5.3 RQ3 - Do the results improve when using a prompt that provides detailed instructions?

For this research question, a prompt was developed using some prompt engineering strategies to achieve even better results. The proposed prompt was structured as follows. First, a persona is defined, specifying the role or function that the model should assume. Next, the objectives are outlined, explaining what needs to be done, how it should be done, and what is expected at the end of the process.

After defining the objectives, a step-by-step guide is provided, detailing the sequence of actions the model should follow to achieve the proposed goals. To assist the model in performing the task, an example of source code is included along with a unit test that verifies whether the code functions correctly. Both the code and the unit test serve as references to be followed.

At the end, additional instructions are given to help the model generate more precise responses. Finally, the source code for which the LLM should generate a unit test is provided, similar to the one used in RQ2. The results can be seen in Table 3.

Table 3: Results presented by LLM models when generating unit tests using a detailed prompt as input

LLM Model	Time	Coverage	Branch Coverage	Hit Rate	Total Price
Gemini 1.5 Flash	2,36	99,85%	99,75%	87,55%	$ 0,0266
Gemini 1.5 Pro	12,08	99,65%	99,52%	89,04%	$ 1,0466
ChatGPT 4o Mini	4,11	99,83%	99,75%	87,03%	$ 0,0463
ChatGPT 4o	6,14	99,59%	99,43%	89,15%	$ 1,3082
HumanEval	–	99,33%	99,06%	96,81%	–

With the use of the detailed prompt, the results for the hit rate metric across all models ranged between 87% and 90%. When compared to the results obtained using the simple prompt combined with the source code (RQ1), this represents a considerable improvement, given that in the latter case, the hit rate values started as low as 72%.

Another noteworthy finding was the slight decrease in hit rates for the Gemini models, both 1.5-pro and 1.5-flash. Since the decrease was small, around 2%, several factors can be taken into consideration. Factors such as response time or the type of prompt used may have negatively impacted these models.

Even though the models do not have a time restriction for responding, the Gemini 1.5-flash model continued to generate responses in about 2 seconds, even when the input size nearly tripled. This fast response time may have negatively impacted its ability to analyze and follow the instructions provided in the detailed prompt.

As for the type of prompt, it is possible that this style is not the most suitable for use with Gemini models, and that more concise prompts with direct instructions could yield better results. However, further studies will be conducted to determine the exact cause of these outcomes.

On the other hand, for the ChatGPT 4o and 4o-mini models, two key points stand out. First, the 4o model showed a slight improvement of around 2%, increasing its hit rate from 87.66% to 89.15%.

The second point to note is that the 4o mini model showed a significant improvement after applying this type of prompt, increasing its hit rate from 75.75% to 87.03%, an improvement of 11.28 percentage points. This brings it much closer to the results achieved by the more expensive model, ChatGPT 4o.

At the end of this stage, it can be concluded that applying the detailed prompt did not lead to a significant improvement in the results of models that already had high hit rates. However, it significantly helped simpler and more affordable models achieve better results, making them much more competitive and financially attractive.

5.4 RQ4 - Which Model Offers the Best Cost-Benefit Ratio for Unit Test Generation?

Finally, an analysis was conducted regarding the pricing for using each model. However, before presenting the costs, it is important to highlight that the values shown refer to one million tokens. A token is a common unit of measurement in natural language processing, representing a word, part of a word, or a character [15]. Therefore, with each request made to an LLM, both the tokens in the input and the tokens in the generated response are counted, determining the final cost.

In terms of cost, Gemini stands out by offering models that are free for users. Both the 1.5-Pro and 1.5-Flash models can be used without cost. However, free usage comes with some restrictions. The 1.5-Flash model allows up to 15 requests per minute, with a limit of 1 million tokens per minute and 1,500 requests per day. The 1.5-Pro model permits up to 2 requests per minute, with a limit of 32,000 tokens and 50 requests per day. On the other hand, OpenAI does not offer a free option for users. Table 4 presents the cost per million tokens for each model.

Table 4: LLM Model Usage Prices

LLM Model	Input	Output
Gemini 1.5 Flash	$ 0,075	$ 0,30
Gemini 1.5 Pro	$ 3,50	$ 10,50
ChatGPT 4o Mini	$ 0,15	$ 0,60
ChatGPT 4o	$ 5,00	$ 15,00

When analyzing Table 4, it becomes clear that the costs associated with using the ChatGPT 4o-mini and Gemini 1.5 Flash models are significantly lower than those of the ChatGPT 4o and Gemini 1.5-Pro models. This makes ChatGPT 4o-mini and Gemini 1.5 Flash much more cost-effective, making them ideal for simpler tasks, given that their accuracy is slightly lower than that of ChatGPT 4o and Gemini 1.5-Pro, as shown in the previous sections.

Next, an analysis was conducted on the costs involved in generating unit tests for the three proposed scenarios: simple prompt, simple prompt with docstring, and detailed prompt. Tables 5, 6, and 7 present these costs.

Table 5: Costs associated with generating unit tests using simple prompt as input

LLM Model	Input Tokens	Input Price	Output Tokens	Output Price	Total Price
Gemini 1.5 Flash	17316	$ 0,0013	39279	$ 0,0118	$ 0,0131
Gemini 1.5 Pro	17316	$ 0,0606	36559	$ 0,3838	$ 0,4444
ChatGPT 4o Mini	16914	$ 0,0025	43073	$ 0,0258	$ 0,0284
ChatGPT 4o	16914	$ 0,0846	49247	$ 0,7387	$ 0,8233

Table 6: Costs associated with generating unit tests using the simple prompt with source code and docstring as input

LLM Model	Input Tokens	Input Price	Output Tokens	Output Price	Total Price
Gemini 1.5 Flash	39302	$ 0,0029	46141	$ 0,0138	$ 0,0168
Gemini 1.5 Pro	39302	$ 0,1376	49922	$ 0,5241	$ 0,6617
ChatGPT 4o Mini	36999	$ 0,0055	38763	$ 0,0233	$ 0,0288
ChatGPT 4o	36999	$ 0,1850	35028	$ 0,5254	$ 0,7104

Table 7: Costs associated with generating unit tests using the detaleid prompt as input

LLM Model	Input Tokens	Input Price	Output Tokens	Output Price	Total Price
Gemini 1.5 Flash	132724	$ 0,0100	55439	$ 0,0166	$ 0,0266
Gemini 1.5 Pro	132724	$ 0,4645	68655	$ 0,7208	$ 1,1853
ChatGPT 4o Mini	120520	$ 0,0181	47042	$ 0,0282	$ 0,0463
ChatGPT 4o	120520	$ 0,6026	53963	$ 0,8094	$ 1,4120

To calculate the costs associated with the activity of unit test generation by the models for the HumanEval dataset, it was necessary to account for the number of tokens identified in both the input (task request) and the output (model response). This distinction was essential because input and output tokens are priced differently.

As shown in Tables 5, 6, and 7, there are significant cost variations between models, especially when considering the different types of inputs provided. Notably, the Gemini 1.5 Flash model demonstrated the best cost-benefit performance, consistently achieving the lowest absolute cost across all three scenarios proposed in this study, even when using the most detailed prompt.

On the other hand, the ChatGPT 4o model exhibited the highest costs, particularly in the scenario involving the detailed prompt, reaching a total of $1.41 to generate tests for all problems in the dataset. Although this model also achieved high hit rates, its elevated cost compared to more affordable alternatives makes it a less viable option in budget-constrained contexts.

The cost gap between the lighter models (Gemini 1.5 Flash and ChatGPT 4o Mini) and the more robust ones (Gemini 1.5 Pro and ChatGPT 4o) was also significant, despite using the same number of input tokens. This reinforces the notion that pricing differences between models have a substantial impact on the economic feasibility of large-scale usage.

Additionally, using the detailed prompt increased the average number of tokens per request, thereby raising total costs. However, this increase proved to be proportionally more advantageous for the cheaper models, such as Gemini 1.5 Flash and ChatGPT 4o Mini, where the additional cost was offset by a significant improvement in accuracy, as discussed in previous sections.

This analysis shows that, when considering cost alone, the Gemini 1.5 Flash model emerges as the most economical option, whereas larger models like GPT-4o, though effective, require a higher investment that may not be justifiable depending on the intended use case.

It is also important to clarify that, since models from the same family (e.g., Gemini or ChatGPT) use the same algorithm for token counting, the number of tokens processed was measured only for the less expensive models and then replicated for their more expensive counterparts. As a result, the input token counts for Gemini 1.5 Flash and Gemini 1.5 Pro are identical, as are those for ChatGPT 4o Mini and ChatGPT 4o. This equivalence does not hold for the output, as each model generated different test code, resulting in varying token counts.

Another factor that contributed to cost reduction was the restriction applied to all prompts, instructing the models to respond exclusively with Python code and omit additional explanations. This constraint had a direct impact on reducing the number of tokens in the output, as the textual descriptions typically included in model responses were suppressed. Since the objective of this study was to evaluate the effectiveness of the generated tests rather than textual justification, this choice enabled tighter control over the volume of processed data and the associated costs.

5.5 Statistical Analysis

In this section, the statistical procedures adopted for analyzing the collected data are described. Initially, the sample and variables were characterized using measures of central tendency (such as mean and median) and dispersion (standard deviation). This was done to identify possible patterns, deviations, or inconsistencies and, thus, gain an overview of the data in order to determine the most appropriate statistical test.

Given that the data may not follow a normal distribution, the Kruskal-Wallis H-test, a non-parametric test, was employed. This test is suitable for comparing three or more independent groups. The test was performed for the main effect of LLM Model, the main effect of Input Type, and the combined effect of LLM Model and Input Type (by creating a Treatment variable). Below are the results for each of the metrics (time, line coverage, branch coverage, and hit rate).

Analysis for Metric: Time

- Kruskal-Wallis H-test for LLM Model (Main Effect): H-statistic = 1267.580, p-value = 0.000. There is a statistically significant difference in Execution Time among the different LLM models.
- Kruskal-Wallis H-test for Input Type (Main Effect): H-statistic = 9.192, p-value = 0.010. There is a statistically significant difference in Execution Time among the Input Types.
- Kruskal-Wallis H-test for Combined Treatment (LLM model x Input Type): H-statistic = 1357.212, p-value = 0.000. There is a statistically significant difference in Execution Time among the combined LLM model x Input treatments.

Analysis for Metric: Line Coverage

- Kruskal-Wallis H-test for LLM Model (Main Effect): H-statistic = 0.278, p-value = 0.964. There is no statistically significant difference in Line Coverage among the LLM Models.
- Kruskal-Wallis H-test for Input Type (Main Effect): H-statistic = 8.746, p-value = 0.013. There is a statistically significant difference in Line Coverage among the Input Types.
- Kruskal-Wallis H-test for Combined Treatment (LLM Model x Input Type): H-statistic = 11.316, p-value = 0.417. There is no statistically significant difference in Line Coverage among the combined LLM Model x Input treatments.

Analysis for Metric: Branch Coverage

- Kruskal-Wallis H-test for LLM Model (Main Effect): H-statistic = 0.445, p-value = 0.931. There is no statistically significant difference in Branch Coverage among the LLM Models.
- Kruskal-Wallis H-test for Input Type (Main Effect): H-statistic = 6.963, p-value = 0.031. There is a statistically significant difference in Branch Coverage among the Input Types.
- Kruskal-Wallis H-test for Combined Treatment (LLM Model x Input Type): H-statistic = 10.116, p-value = 0.520. There is no statistically significant difference in Branch Coverage among the combined LLM Model x Input treatments.

Analysis for Metric: Hit Rate

- Kruskal-Wallis H-test for LLM Model (Main Effect): H-statistic = 22.177, p-value = 0.000. There is a statistically significant difference in Hit Rate among the LLM Models.
- Kruskal-Wallis H-test for Input Type (Main Effect): H-statistic = 44.136, p-value = 0.000. There is a statistically significant difference in Hit Rate among the Input Types.
- Kruskal-Wallis H-test for Combined Treatment (LLM Model x Input Type): H-statistic = 83.732, p-value = 0.000. There is a statistically significant difference in Hit Rate among the combined LLM Model x Input treatments.

The results indicate that both the LLM model and the Input Type (as well as their combinations) have a statistically significant impact on execution time. Visually, Gemini 1.5 Flash consistently shows the lowest execution times across different input types.

For the coverage metrics (line and branch), the LLM models themselves did not show statistically significant differences. However, the Input Type had a significant impact on both Line Coverage and Branch Coverage. This suggests that the way the data is provided (e.g., source code, source code + docstring, detailed prompt) is more relevant to the coverage achieved than the specific LLM model used among those tested. Based on descriptive statistics, "source code + docstring" and "detailed prompt" generally lead to higher average/median coverage.

Finally, the hit rate metric also showed statistically significant differences regarding the LLM model used, the Input Type, and their combinations.

From this statistical analysis, it is concluded that no single LLM model is ideal across all metrics. For instance, one model may be faster but have higher costs or slightly lower coverage.

6 Limitations and Threats to Validity

Despite the contributions of this study, there are some limitations that will be addressed in future work. This section presents the limitations and threats to validity identified in this work, organized into four dimensions: internal validity, external validity, construct validity, and conclusion validity, in line with Kitchenham's guidelines [29].

Regarding internal validity, the evaluated models were designed for general-purpose applications, meaning they were not specifically trained for code-related tasks. While this reflects a realistic use of publicly available tools, it may limit the accuracy of the results. It is likely that models such as Codex, which were designed with a programming focus, could achieve better performance in this context. In future work, techniques such as fine-tuning [37] [25] will be explored to enhance model specialization.

Another potential threat to the internal validity of this study is data leakage, which occurs when the examples used in the experiments are partially or entirely included in the training data of the evaluated language models. This may happen because the dataset was published in 2020 or due to its widespread use in other studies.

With respect to external validity, this study focused on four LLMs developed by two different companies. These models were selected due to their relative novelty and the fact that they have not yet been extensively studied. Although GPT-based models have been widely analyzed in the literature, recent versions such as GPT-4o and GPT-4o-mini are not yet frequently explored. However, other models, such as those from Anthropic and the LLaMA family, could also have been included to broaden the analysis.

Also related to external validity, the experiment was conducted exclusively with the Python programming language. Although Python is widely adopted in both academia and industry, the results cannot be generalized to other programming languages, such as Java, C++, or JavaScript, which differ in syntax and semantics.

Regarding construct validity, it is worth noting that the selected dataset effectively served the purpose of this study, which was to compare model efficiency, particularly in terms of hit rate and cost. However, it is composed of relatively simple problems. For a more comprehensive analysis, it would be necessary to employ datasets with more complex problems or even real-world software projects available in public repositories such as GitHub.

Another point to note concerning construct validity is that in real-world projects, the process of automatically generating unit tests presents additional challenges, such as inter-module dependencies and the need for mock objects. Since the dataset used contains only low-complexity problems, these challenges were not addressed in this study.

Finally, with respect to conclusion validity, only functional metrics were used in this study, such as line coverage, branch coverage, and hit rate. Important aspects such as readability, maintainability, and complexity of the generated tests were not considered. These elements are essential for assessing the practical utility of tests in real-world development environments and will be addressed in future research.

7 Conclusion

This study conducted a detailed analysis of the effectiveness of various LLM models from Google and OpenAI in the automatic generation of unit tests. Models ranging from the simplest to the most advanced from these companies were evaluated based on their ability to generate unit tests from the source code in the HumanEval dataset. The generated tests were assessed using the criteria of line coverage, branch coverage, hit rate, and cost.

The tests were generated using three different types of input. Initially, only the source code of the method to be tested was provided. In the second phase, the input included both the source code and a docstring. Finally, a detailed prompt outlining the steps the models should follow to perform the task was provided as input.

The results showed that all LLM models generate high-quality unit tests, with line and branch coverage rates close to 100%. However, hit rates vary depending on the model used and the type of input provided. These rates range from 72.61% for the GPT-4o-mini model when given only the method's source code as input to 90.69% for the Gemini 1.5-pro model when provided with the source code along with comments. This is a value very close to the hit rate achieved by the tests contained in the dataset, which reached 96.81%.

The costs involved also vary depending on the model used and the type of input provided. The highest cost recorded was for the GPT-4o model, which required $1.3082 to generate tests for all problems in the dataset when given the detailed prompt as input, achieving a hit rate of 89.15%. However, it is possible to obtain better results at a lower cost. For instance, the Gemini 1.5-flash model achieved a hit rate of 89.77% at a cost of just $0.0168.

REFERENCES

[1] S. S. Riaz Ahamed. 2010. Studying the Feasibility and Importance of Software Testing: An Analysis. arXiv:1001.4193 [cs.SE] https://arxiv.org/abs/1001.4193

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[4] K. Beck. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley. https://books.google.com.br/books?id=G8EL4H4vf7UC

[5] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools. arXiv:2312.10622 [cs.SE] https://arxiv.org/abs/2312.10622

[6] Sonam Bhatia and Jyoteesh Malhotra. 2014. A survey on impact of lines of code on software complexity. In *2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014)*. 1–4. doi:10.1109/ICAETR.2014.7012875

[7] Igor B Bourdonov, Alexander S Kossatchev, Victor V Kuliamin, and Alexander K Petrenko. 2002. UniTesK test suite architecture. In *FME 2002: Formal Methods—Getting IT Right: International Symposium of Formal Methods Europe Copenhagen, Denmark, July 22–24, 2002 Proceedings*. Springer, 77–88.

[8] Denivan Campos, Luana Martins, and Ivan Machado. 2022. An empirical study on the influence of developers' experience on software test code quality. In *Anais do XXI Simpósio Brasileiro de Qualidade de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 18–27. https://sol.sbc.org.br/index.php/sbqs/article/view/23288

[9] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235. doi:10.1016/j.infsof.2018.08.010

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf,

Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[11] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211. doi:10.1109/ISSRE.2014.11

[12] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology* 171 (2024), 107468. doi:10.1016/j.infsof.2024.107468

[13] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic Complexity. *IEEE Software* 33, 6 (2016), 27–29. doi:10.1109/MS.2016.147

[14] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically evaluating readily available information for regression test optimization in continuous integration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 491–504. doi:10.1145/3460319.3464834

[15] Ali Erkan and Tunga Güngör. 2023. Analysis of Deep Learning Model Combinations and Tokenization Approaches in Sentiment Classification. *IEEE Access* 11 (2023), 134951–134968. doi:10.1109/ACCESS.2023.3337354

[16] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *International Conference On Quality Software (QSIC)*. IEEE Computer Society, Los Alamitos, CA, USA, 31–40. doi:10.1109/QSIC.2011.19

[17] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[18] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. doi:10.1109/TSE.2012.14

[19] Yi Gao, Xing Hu, Zirui Chen, Xiaohu Yang, and Xin Xia. 2024. Unit Test Generation for Vulnerability Exploitation in Java Third-Party Libraries. arXiv:2409.16701 [cs.SE] https://arxiv.org/abs/2409.16701

[20] Danielle Gonzalez, Joanna C.S. Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. 2017. A Large-Scale Study on the Usage of Testing Patterns That Address Maintainability Attributes: Patterns for Ease of Modification, Diagnoses, and Comprehension. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 391–401. doi:10.1109/MSR.2017.8

[21] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[22] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. 2018. An Empirical Investigation on the Readability of Manual and Generated Test Cases. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 348–3483.

[23] Jay Graylin, Randy K SMITH, HALE David, Nicholas A KRAFT, WARD Charles, et al. 2009. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications* 2, 3 (2009), 137–143.

[24] Paul Hamill. 2004. *Unit test frameworks: tools for high-quality software development*. " O'Reilly Media, Inc.".

[25] Yi Hu, Hyeonjin Kim, Kai Ye, and Ning Lu. 2025. Applying fine-tuned LLMs for reducing data needs in load profile analysis. *Applied Energy* 377 (2025), 124666. doi:10.1016/j.apenergy.2024.124666

[26] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 319–321. doi:10.1145/3510454.3522684

[27] Shujuan Jiang, Miao Zhang, Yanmei Zhang, Rongcun Wang, Qiao Yu, and Jacky Wai Keung. 2021. An Integration Test Order Strategy to Consider Control Coupling. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1350–1367. doi:10.1109/TSE.2019.2921965

[28] Rabimba Karanjai, Aftab Hussain, Md Rafiqul Islam Rabin, Lei Xu, Weidong Shi, and Mohammad Amin Alipour. 2024. Harnessing the Power of LLMs: Automating Unit Test Generation for High-Performance Computing. arXiv:2407.05202 [cs.SE] https://arxiv.org/abs/2407.05202

[29] Barbara Kitchenham. 2004. *Procedures for Performing Systematic Reviews*. Technical Report TR/SE-0401. Keele University, Keele, UK.

[30] Roberto Latorre. 2013. Effects of developer experience on learning and applying unit test-driven development. *IEEE Transactions on Software Engineering* 40, 4 (2013), 381–395.

[31] Werney Lira, Pedro Santos Neto, and Luiz Osorio. 2024. Uma análise do uso de ferramentas de geração de código por alunos de computação. In *Anais do IV Simpósio Brasileiro de Educação em Computação* (Evento Online). SBC, Porto Alegre, RS, Brasil, 63–71. doi:10.5753/educomp.2024.237427

[32] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. 2024. A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites. arXiv:2408.07846 [cs.SE] https://arxiv.org/abs/2408.07846

[33] Luiz Osorio, Pedro Santos Neto, Guilherme Avelino, and Werney Lira. 2025. An Evaluation of the Impact of Code Generation Tools on Software Development. In *Anais do XXI Simpósio Brasileiro de Sistemas de Informação* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 625–634. doi:10.5753/sbsi.2025.246605

[34] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. 2024. Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation. arXiv:2407.00225 [cs.SE] https://arxiv.org/abs/2407.00225

[35] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM.

[36] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2024. Multi-language Unit Test Generation using LLMs. arXiv:2409.03093 [cs.SE] https://arxiv.org/abs/2409.03093

[37] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2024. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology* 175 (2024), 107523. doi:10.1016/j.infsof.2024.107523

[38] R.S. Pressman. 2005. *Software Engineering: A Practitioner's Approach*. Boston. https://books.google.com.br/books?id=bL7QZHtWvaUC

[39] Rudolf Ramler, Dietmar Winkler, and Martina Schmidt. 2012. Random Test Case Generation and Manual Unit Testing: Substitute or Complement in Retrofitting Tests for Legacy Code?. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. 286–293. doi:10.1109/SEAA.2012.42

[40] Linda Rosenberg, Ted Hammer, and Jack Shaw. 1998. Software metrics and reliability. In *9th international symposium on software reliability engineering*.

[41] Shexmo Santos, Raiane Fernandes, Marcos Santos, Michel Soares, Fabio Rocha, and Sabrina Marczak. 2024. Increasing Test Coverage by Automating BDD Tests in Proofs of Concepts (POCs) using LLM. In *Anais do XXIII Simpósio Brasileiro de Qualidade de Software* (Bahia/BA). SBC, Porto Alegre, RS, Brasil, 519–525. https://sol.sbc.org.br/index.php/sbqs/article/view/32979

[42] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[43] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 121–125. doi:10.1109/MSR.2019.00028

[44] J. Shore and S. Warden. 2021. *The Art of Agile Development* (2nd ed.). O'Reilly Media, Incorporated.

[45] Daniil Stepanov and Dmitry Ivanov. 2023. Automated Unit Test Generation For Java Programs Using Fuzzing. In *2023 Ivannikov Ispras Open Conference (ISPRAS)*. 157–162. doi:10.1109/ISPRAS60948.2023.10508168

[46] Ye Tian, Beibei Yin, and Chenglong Li. 2021. A Model-based Test Cases Generation Method for Spacecraft Software. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*. 373–382. doi:10.1109/DSA52907.2021.00057

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[48] Han Wang, Han Hu, Chunyang Chen, and Burak Turhan. 2024. Chat-like Asserts Prediction with the Support of Large Language Model. arXiv:2407.21429 [cs.SE] https://arxiv.org/abs/2407.21429

[49] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. arXiv:2408.11324 [cs.SE] https://arxiv.org/abs/2408.11324

[50] Qinyuan Ye, Maxamed Axmed, Reid Pryzant, and Fereshte Khani. 2024. Prompt Engineering a Prompt Engineer. arXiv:2311.05661 [cs.CL] https://arxiv.org/abs/2311.05661

[51] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 76 (July 2024), 24 pages. doi:10.1145/3660783

[52] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2024. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv:2305.04207 [cs.SE] https://arxiv.org/abs/2305.04207