An initial investigation of ChatGPT unit test generation capability[*]

Vitor H. Guilherme[†]
vitor.guilherme@estudante.ufscar.br
Federal University of São Carlos
São Carlos, SP, Brazil

Auri M. R. Vincenzi[†]
auri@ufscar.br
Federal University of São Carlos
São Carlos, SP, Brazil

ABSTRACT

Context: Software testing ensures software quality, but developers often disregard it. The use of automated testing generation is pursued to reduce the consequences of overlooked test cases in a software project. **Problem**: In the context of Java programs, several tools can completely automate generating unit test sets. Additionally, studies are conducted to offer evidence regarding the quality of the generated test sets. However, it is worth noting that these tools rely on machine learning and other AI algorithms rather than incorporating the latest advancements in Large Language Models (LLMs). **Solution**: This work aims to evaluate the quality of Java unit tests generated by an OpenAI LLM algorithm, using metrics like code coverage and mutation test score. **Method**: For this study, 33 programs used by other researchers in the field of automated test generation were selected. This approach was employed to establish a baseline for comparison purposes. For each program, 33 unit test sets were generated automatically, without human interference, by changing Open AI API parameters. After executing each test set, metrics such as code line coverage, mutation score, and success rate of test execution were collected to evaluate the efficiency and effectiveness of each set. **Summary of Results**: Our findings revealed that the OpenAI LLM test set demonstrated similar performance across all evaluated aspects compared to traditional automated Java test generation tools used in the previous research. These results are particularly remarkable considering the simplicity of the experiment and the fact that the generated test code did not undergo human analysis.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Empirical software validation**; **Software defect analysis**.

KEYWORDS

software testing, experimental software engineering, automated test generation, coverage testing, mutation testing, testing tools

[*]This work is partially supported by Brazilian Funding Agencies CAPES - Grant 001, FAPESP - Grant nº 2019/23160-0, and CNPq.

[†]All authors contributed equally to the paper content.

ACM Reference Format:
Vitor H. Guilherme[†] and Auri M. R. Vincenzi[†]. 2023. An initial investigation of ChatGPT unit test generation capability[*]. In *8th Brazilian Symposium on Systematic and Automated Software Testing (SAST 2023), September 25–29, 2023, Campo Grande, MS, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624032.3624035

1 INTRODUCTION

Unit testing is an essential practice in software development to ensure the correctness and robustness of individual code units. These tests, typically written by developers, play a crucial role in identifying defects and validating the expected behavior of software components. DevOps pipelines are strongly based on the quality of the unit tests. However, manually generating comprehensive unit tests can be challenging and time-consuming, often requiring significant effort and expertise. To address these challenges, researchers have explored automated approaches for test generation [1, 8], leveraging advanced techniques and tools.

In this work, we focus on evaluating the quality of Java unit tests generated by an OpenAI Large Language Model (LLM) that has demonstrated remarkable capabilities in generating tests across various domains [23, 28, 29]. Our evaluation will utilize three key quality parameters: code coverage, mutation score, and build and execute success rate of test sets. Code coverage quantifies the extent to which the tests exercise different parts of the code, indicating the thoroughness of the test suite. On the other hand, the mutation score measures the ability of the tests to detect and kill mutated versions of the code, providing insights into the fault-detection capability [2]. Finally, the build and success execution rate measures the reliability of the generated tests.

To conduct a thorough and comprehensive analysis, this study will compare the quality of the unit tests generated by the selected LLM with those produced by other prominent Java test generation tools, such as EvoSuite[1]. This comparative evaluation aims to determine if the LLMs can outperform state-of-the-art Java test generation tools and will leverage relevant data from Araujo and Vincenzi [3] research to provide a meaningful benchmark for comparison. By assessing the effectiveness and performance of the LLMs against established tools, we can gain valuable insights into their capabilities and potential advantages in generating high-quality unit tests for Java programs.

This study is part of an ongoing project that aims to support mutation testing in a fully automated way[2]. In this sense, we are

[1]https://www.evosuite.org/
[2]Mutation-Based Software Testing with High Efficiency and Low Technical Debt: Automated Process and Free Support Environment Prototype (FAPESP Grant Nº 2019/23160-0)

investigating tools that allow us to generate test cases in a fully automated way without human intervention/interaction.

Therefore, we can summarize these paper's contributions:

- To provide evidence of the quality of LLMs in generating unit test sets for Java programs concerning their efficiency and efficacy;
- To evaluate the improvements a combination of test sets can achieve over individual test sets concerning efficiency and efficacy;
- To collect data for supporting further comparison of different LLMs on generating Java unit test sets;
- To develop and make available a set of artifacts for easing the experimentation for different sets of programs.

The structure of the rest of this paper is as follows: We outline the essential subjects for comprehension of this paper in Section 2. In Section 3, we touch on other studies that are related to ours and highlight the differences. The design of our experiment, along with our choices of programs and tools, is detailed in Section 4. Section 5 displays the data we've gathered and the subsequent analysis. A discussion of the outcomes derived from the collected data is provided in Section 6. We then discuss potential risks that may affect our experiment results in Section 7. Finally, in Section 8, we wrap up the paper by indicating possible future research directions informed by this study and the data collected.

2 BACKGROUND

This section will explain software testing, automatic test data generation, and large language models so that the rest of the paper can be understood.

2.1 Software testing

In the sphere of software development, it is crucial to ensure the robustness and reliability of a program. A primary technique used for this goal is software testing, which is a systematic process that checks the functionality and accuracy of a software application. However, with software systems growing increasingly complex and versatile, covering a broadening range of use cases and inputs makes software testing an arduous task.

To analyze the effectiveness of a test set, various criteria come into play, two of which are lines of code coverage and mutation testing[20]. Code coverage entails analyzing the extent to which the test suite exercises the internal structure of the software product, like its statements or conditions. The goal is to achieve complete or near-complete coverage to ensure that each statement or branch in the code has been executed at least once for a given test case during testing.

On the other hand, mutation testing evaluates the test suite's ability to identify and "kill" mutated versions of the software [7]. Mutation testing can be seen as a fault model representation [2]. These mutations involve making small syntax changes to the code to simulate potential faults. A successful mutation test is one in which the test suite effectively detects these mutations, highlighting its proficiency in identifying vulnerabilities and potential issues within the software. Both code coverage and mutation testing are used in this study as metrics to measure the reliability and thoroughness of the automatically generated test sets. Moreover, these are traditional metrics used in other studies, like the one developed by Araujo and Vincenzi [3], which we will use as a baseline.

2.2 Traditional automatic test data generation

The automatic generation of test data poses an undecidable problem from a computational perspective. While random testing or search-based strategies are commonly employed, other research has shown that the problem remains unsolved when using traditional tools that rely on these approaches [3, 25]. The shortcomings of traditional test data generators become apparent when attempting to achieve all testing objectives, such as complete code coverage or eliminating all mutants [1]. Consequently, pursuing comprehensive and efficient test data generation techniques continues to be an ongoing challenge in the dynamic field of software testing.

Even considering the state-of-the-art unit testing generation for Java, EvoSuite[9], the resultant test set reaches low mutation scores in traditional competitions [22, 26]. Other tools have been discontinued, like Palus [30] and JTExpert [21], which also employed search-based algorithms. There are also tools that employ a random generation approach like Randoop [18] but are still being updated.

2.3 LLM and Software Engineering

LLMs, like ChatGPT[3], are state-of-the-art language models based on the Transformer architecture [24]. They are designed to process and understand human language, enabling machines to generate coherent and contextually relevant text. These models have been trained on vast amounts of language data, allowing them to capture intricate patterns and relationships in language usage. As a result, they demonstrate impressive capabilities in tasks such as text generation, translation, question-answering, and even software-related activities.

Ma et al. [16] comprehensively explore ChatGPT's applicability and potential in the software engineering field. The authors examine various tasks, including code generation, code summarizing, bug detection, and code completion, to evaluate the performance of ChatGPT. Through a rigorous investigation and comparison with existing software engineering tools and techniques, the study reveals both the strengths and limitations of ChatGPT in different software engineering scenarios. The findings provide valuable insights into the capabilities of ChatGPT and offer guidance for leveraging its potential to improve software development practices while highlighting areas where further advancements are needed.

White et al. [27] also explore the potential applications of ChatGPT in various software engineering tasks. The researchers introduce a collection of prompt patterns specifically designed to leverage ChatGPT's language generation capabilities for code quality improvement, refactoring, requirements elicitation, and software design tasks. Through experiments and case studies, they demonstrate the effectiveness of using ChatGPT with these prompt patterns in aiding developers and software engineers in their day-to-day activities. The article highlights the versatility of ChatGPT as a tool for supporting software engineering practices and fostering better code development and design.

[3]https://chat.openai.com/

2.4 LLM for automatic test data generation

Section 3 explores the possibility of leveraging LLMs for automatic test data generation. These studies involved exploratory investigations into using LLMs to generate test data across different testing phases, from unit to end-to-end testing. Notably, the context provided to the LLM was the only aspect that changed during these experiments.

In the case of unit testing, the LLM was presented with code snippets as input [14, 23, 28, 29]. For instance, a prompt could be formulated as follows:

"Given the code snippet provided, please generate test cases to cover all possible scenarios and branches within the code."

The LLM then utilized its language generation capabilities to produce comprehensive test data sets that catered to various testing scenarios.

On the other hand, for end-to-end testing, the LLM was supplied with a description of the system's functional specifications [19] or a GUI [15]. The prompt may have asked the LLM to:

"Generate test cases that validate the entire system's functionality based on the provided functional specification."

The results of these exploratory studies demonstrated the promising potential of LLMs in automating the test data generation process, streamlining testing efforts, and enhancing software quality. By tailoring the input context to the LLM's capabilities, it was possible to obtain effective test cases for different testing phases, further showcasing the versatility and adaptability of LLMs in software testing.

3 RELATED WORK

The field of test generation has witnessed significant advancements in recent years, with researchers exploring innovative approaches to automate the process and enhance software quality assurance practices. Among these emerging techniques, one notable area of exploration is using LLMs for test generation. This section provides a comprehensive overview of the existing literature investigating the application of these powerful tools in test generation.

Li et al. [14] introduce a novel approach to detecting failure-inducing test cases using ChatGPT. By leveraging the model's ability to understand natural language and generate coherent responses, they propose an interactive debugging technique that allows developers to converse with ChatGPT to identify test cases that are likely to trigger failures. Through experiments on real-world software projects, they demonstrate the effectiveness of their approach in improving fault localization and aiding in the debugging process, highlighting its potential to enhance quality assurance practices.

Yuan et al. [29] explore the application of ChatGPT for automating unit test generation. The researchers evaluate the performance of ChatGPT in generating meaningful and effective unit tests by comparing them with existing test-generation tools. They also propose a novel approach to enhance ChatGPT's ability to generate high-quality unit tests by incorporating reinforcement learning techniques. Through rigorous experimentation and evaluation of various code bases, the authors demonstrate the potential of ChatGPT as a promising tool for automating the labor-intensive task of unit test generation, highlighting its ability to improve software testing efficiency and accuracy.

Siddiq et al. [23] investigate the efficacy of large language models, specifically GPT-3, in generating unit tests for software programs. The study explores the ability of GPT-3 to understand the requirements of software functionalities and generate relevant test cases. The author analyzes the quality, diversity, and coverage of the generated unit tests through experiments conducted on real-world projects, comparing them with manually written tests. The findings highlight the potential of large language models in automated unit test generation but also reveal certain limitations and challenges that need to be addressed for more effective and reliable results. The research contributes to understanding the capabilities and limitations of large language models in the context of unit testing and provides insights for further advancements in this area.

Xie et al. [28] present ChatUniTest, a tool that allows developers to interact with ChatGPT in a conversational manner to generate unit tests for their code. By formulating test generation as a dialogue-based problem, developers can provide natural language prompts to ChatGPT, which then responds with relevant test case suggestions. The article discusses the implementation details of ChatUniTest and evaluates its effectiveness through experiments on open-source projects. The results demonstrate that ChatUniTest successfully generates meaningful unit tests, assisting developers in improving software quality and productivity. The study highlights the potential of ChatGPT in the context of automated unit test generation and presents an innovative approach for facilitating the software testing process.

Liu et al. [15] explore the application of GPT-3 for automated GUI testing in the context of mobile applications. The study proposes an innovative approach where GPT-3 is utilized as a conversational agent to interact with mobile apps and generate test cases. A series of experiments conducted on various real-world mobile apps demonstrate the feasibility of GPT-3 in performing human-like GUI testing. The approach achieves high code coverage and successfully detects critical issues, showcasing the potential of leveraging GPT-3 for efficient and effective automated GUI testing of mobile applications. The findings highlight the capabilities of GPT-3 in the domain of mobile app testing, opening avenues for further advancements in automated testing techniques.

Considering the studies carried out so far, the majority explore the use of ChatGPT in an interactive way. We intend to investigate the ChatGPT test generation capability fully automated, without human intervention, interacting, or correcting test cases, considering a possible scenario of no-touch testing [1, 8]. In this sense, we consider our study incomparable to the previous ones.

4 EXPERIMENT DESIGN

This paper evaluates the quality of automatically generated test sets by an LLM. A set of Java programs was carefully selected to accomplish this, and multiple JUnit[4] test sets were generated using the LLM. We use the OpenAI API[5] (Application Programming Interface) and develop a Python script for interacting with the model via API. The generated test sets will be evaluated based on code coverage, mutation score, and build and execution success rate using selected tools.

[4]https://junit.org/
[5]https://openai.com/blog/openai-api

The collected data will be summarized and analyzed using simple statistics to compare test sets generated by the LLMs with the ones generated by other automated test-generation tools. Figure 1 illustrates the experiment workflow and the steps involved in the evaluation process.

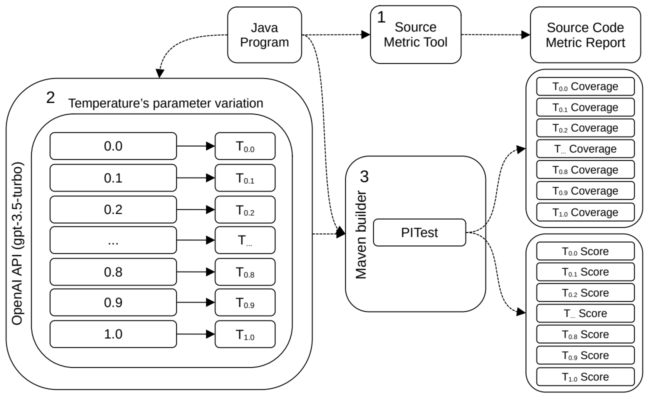

Figure 1: Experiment Design Diagram

Each dashed arrow indicates the input for the subsequent step. Initially, in the first step, we compute some static metrics from the Java source code using JavaNCSS[6] metric tool.

In the second step, we provide the Python script with a personalized prompt, a program under testing, and a "temperature" parameter (see Section 4.1), considering the OpenAI API `gpt-3.5-turbo` model. This step results in the generation of 33 test sets per program, 3 for each temperature.

Subsequently, a program and its test sets are submitted to the PITest [6] tool in the third step. The PITest tool generates all its mutants for the program under testing and executes each test set against its set of mutants, producing a comprehensive report that includes the mutation score and code coverage for each test set. Following, we comment on some decisions for experiment execution.

4.1 LLM selection

The field of large language models has witnessed remarkable progress, with new ones being developed almost daily. OpenAI, one of the leading organizations in the domain, has been at the forefront of LLM development. In this paper, we choose to leverage the power of OpenAI's `gpt-3.5-turbo` model due to its availability as a free model and its association with ChatGPT, making it the most used model by final users.

It is worth mentioning that OpenAI had previously introduced a code generation-focused model named `davinci-code`. However, this model has been discontinued, making `gpt-3.5-turbo` the preferable option for code-related tasks in our study [17].

An important thing about OpenAI API is the temperature parameter. It is a feature that allows users to control the level of randomness and creativity in the generated text. The API can adjust the temperature value to influence the output's diversity and exploration. Higher temperatures, such as 1.0, encourage more randomness in the generated text, resulting in imaginative and varied responses. On the other hand, a lower temperature, like 0.2, produces more focused and deterministic output, favoring predictable and conservative responses. By adjusting the temperature parameter, users can fine-tune the balance between generating creative and coherent text, enabling them to obtain the desired output level for their specific application or task.

We conducted the experiment using the range of temperature values to investigate the variation in the results we will obtain. By exploring the entire spectrum of available temperature values, we aimed to identify the most suitable setting to yield the best results for our specific test generation requirements. Because of the randomness of the model, especially with higher temperature values, we generated 3 test sets for each temperature value and used the average results to minimize bias.

4.2 Program Selection

We used the results from Araujo and Vincenzi [3]'s work as a baseline. Araujo and Vincenzi [3] used a set of 33 Java programs and conducted an experiment investigating the capability of four different automatic testing generators (EvoSuite [11], Palus [30], Randoop [18], and JTExpert [21]) for Java on covering code and killing mutants using PITest [6] as the mutation tool.

Therefore, we selected the same set of programs to perform our experiment. We make all the programs and scripts available at our GitHub repository[7]. By comparing the test sets produced by GPT-Turbo-3.5 with those generated by these tools, our research aims to provide valuable insights into the effectiveness and efficacy of LLMs in automated unit test generation. Table 1 shows the selected programs and their characteristics.

For each program Araujo and Vincenzi [3] computed the following metrics:

- Non-Commenting Source Statements (NCSS);
- Cyclomatic Complexity Number (CCN);
- Cyclomatic Complexity Average (CCA);
- Number of requirements demanded to cover statement coverage; and
- Number of generated mutants considering all mutation operators available in PIT.

As can be observed, they are not complex programs but once we are working at unit testing levels, we understand that each program provides units with sufficient complexity, equivalent to units present in other real programs. Regarding lines of code, the average size is around 40, and cyclomatic complexity is around 4.9.

We operate under the assumption that all programs adhere to their specifications. Consequently, any test case that runs successfully on these programs is deemed correct concerning the program specification, thus sidestepping the oracle problem [5]. This assumption seems reasonable given that several automated testing generators, such as EvoSuite [11], Palus [30], Randoop [18], and JTExpert [21] for Java, generate test cases that pass in the existing product implementation, meaning no test case will fail. The creators of these tools refer to these test cases, which treat the current output as the expected output, as "regression test cases" [18] because they are useful to validate future changes on the current implementation.

[6]https://javancss.github.io/

[7]https://github.com/aurimrv/initial-investigation-chatgpt-unit-tests.git

Table 1: Static information of the Java programs (extracted from Araujo and Vincenzi [3])

ID	Program	#Classes	#Methods	NCSS	CCN	CCA	#Req	#Mut
1	Max	1	1	8	3	3,0	4	14
2	MaxMin1	1	1	13	4	4,0	8	21
3	MaxMin2	1	1	14	4	4,0	8	21
4	MaxMin3	1	1	32	9	9,0	16	61
5	Sort1	1	1	11	4	4,0	10	21
6	FibRec	1	1	8	2	2,0	6	12
7	FibIte	1	1	8	2	2,0	6	12
8	MaxMinRec	1	1	26	5	5,0	13	41
9	Mergesort	1	2	22	6	4,0	16	56
10	MultMatrixCost	1	1	18	6	6,0	14	75
11	ListArray	1	4	20	3	1,8	12	29
12	ListAutoRef	2	4	23	2	1,3	12	21
13	StackArray	1	5	20	3	1,8	12	27
14	StackAutoRef	2	5	27	3	1,4	17	27
15	QueueArray	1	5	24	3	2,0	19	40
16	QueueAutoRef	2	5	32	3	1,6	23	32
17	Sort2	2	7	74	6	3,4	49	141
18	HeapSort	1	9	59	5	2,7	40	116
19	PartialSorting	1	10	62	5	2,5	42	120
20	BinarySearch	1	4	32	8	3,5	21	55
21	BinaryTree	2	11	85	7	3,0	48	145
22	Hashing1	2	10	61	5	2,1	35	88
23	Hashing2	2	12	88	7	3,2	51	162
24	GraphMatAdj	1	9	60	5	2,9	42	134
25	GraphListAdj1	3	16	66	4	1,6	34	95
26	GraphListAdj2	2	14	88	6	2,2	51	113
27	DepthFirstSearch	3	16	65	4	1,6	33	94
28	BreadthFirstSearch	3	16	65	4	1,6	33	94
29	Graph	3	16	65	4	1,6	33	94
30	PrimAlg	1	5	40	7	2,6	31	71
31	ExactMatch	1	4	55	8	6,3	40	205
32	AproximateMatch	1	1	24	7	7,0	19	88
33	Identifier	1	3	30	9	7,7	22	114
Avg		1,5	6,1	40,2	4,9	3,3	24,8	73,9
SD		0,7	5,2	25,4	2,0	2,0	14,7	50,3

4.3 Tools Selection

We used JUnit as a unit testing framework to evaluate the results, which is widely recognized as the industry standard for testing and generating comprehensive reports. Another noteworthy aspect is the utilization of JUnit in the study of Araujo and Vincenzi [3], which is a valuable reference point for comparing our results.

By employing the same testing framework, we establish a meaningful basis for comparison, enabling us to analyze and assess the effectiveness of our LLM-generated tests concerning their findings. The same logic was used to select PITest[8] as our mutation tool. Same as Araujo and Vincenzi [3], we employed all the mutation operators of PITest[9] to generate as many mutants as possible for each program under testing.

Therefore, Table 2 summarizes the tools and versions we used, which we kept the same as the ones adopted by Araujo and Vincenzi [3] to minimize threats.

5 DATA COLLECTION AND ANALYSIS

The initial step involved creating a centralized repository housing all the selected programs, scripts, and experimental results. To achieve version control and facilitate seamless collaboration, we opted for GitHub as our hosting platform[10].

[8]https://pitest.org/
[9]https://pitest.org/quickstart/mutators/
[10]https://github.com/aurimrv/initial-investigation-chatgpt-unit-tests.git

Table 2: Tools version and purpose (adapted from Araujo and Vincenzi [3])

Tool	Version	Purpose
JavaNCSS	32.53	Static Metric Computation
PITest	1.3.2	Mutation and Coverage Testing
Maven	3.6.3	Application Builder
JUnit	4.12	Framework for Unit Testing
Python	3.7	Script language
Java	8	Programs language
LLM	gpt-3.5-turbo	OpenAI LLM for generating tests

Once we use the programs from Araujo and Vincenzi [3] we simplest use the static metrics from their work without recomputing them. Table 1 presented such data about the programs.

Subsequently, we proceeded with the test set generation using the `gpt-3.5-turbo` model. To accomplish this, we formulated a specific base prompt designed to request the model's assistance in generating JUnit unit tests tailored for a program.

The first prompt version is as follows:

In Figure 2, `{cut}` represents the class's name under testing, and `{code}` is a variable containing the CUT source and its dependencies. We developed a Python script (`generate-chatgpt.py`), which sends the request to the OpenAI API. Upon receiving the response from the API, the script removes any natural language comments that the LLM model added before or after the generated code. Additionally, the script ensured that the Java test class

```
Generate test cases just for the {cut}
Java class in one Java class file with
imports using JUnit 4 and Java 8:

{code}
```

Figure 2: Prompt version 1 for test set generation

name matched the file name, following a pattern to enhance test data organization. As an output, the script generates 33 Java test classes for every selected program, with 3 test classes for each LLM temperature value, as mentioned in Section 4.1.

Then, with all tests generated for every program, it was time to build and run them using Maven and PITest. To automate this process, we developed another Python script (`compile-and-test-chatgpt.py`). However, at this stage, we encountered an issue where some tests generated by the model didn't build successfully due to problems such as syntax errors and missing imports. The script simplest discards any test set with failing cases once it is only possible to call PITest after a successful build and test execution. The script moves all test files to a directory outside the project, copies one test file at a time to the project's test directory, and then builds and runs the test for that specific file. This way, any build issues or errors in one test won't affect the others, ensuring a smoother and more effective testing process.

Finally, we developed the last Python script (`reports-chatgpt.py`) for extracting coverage and mutation scores from PITest reports. It is responsible for generating one CSV file for each Java program, including all test results that are executed successfully. Tables 3 and 4 present parts of the collected data. Considering the first prompt version, presented in Figure 2, Table 3 presents average data for each temperature value we investigate.

Table 3: Average Data for Each Temperature Parameter – Prompt version 1

Temp.	# of Suc. Test	% of Suc.	AVG Cov.	AVG Score
0.0	37	37.4	83.0	51.3
0.1	37	37.4	85.7	51.6
0.2	37	37.4	84.6	52.3
0.3	38	38.4	86.1	53.9
0.4	39	39.4	86.9	53.4
0.5	35	35.4	88.3	53.6
0.6	52	52.5	83.9	54.4
0.7	36	36.4	88.9	54.8
0.8	45	45.5	87.6	54.2
0.9	42	42.4	81.5	49.2
1.0	41	41.4	81.8	52.9

Observe that the average results in Table 3 show that from a possible total of 99 test sets for each temperature (3 for each of 33 programs), the temperature most effective on generating successful test sets is 0.6. With this temperature, 52 out of 99 test sets run correctly with no errors, with a successful rate of 52.5%.

Table 3 also shows that quantity does not mean high-quality tests. Temperature 0.7 reaches 36.4 of the successful rate of test sets, around 16% less than temperature 0.6, but with 36 test sets, the average coverage and mutation score are the highest: 88.9 and 54.8, respectively. Although we consider these results impressive due to the simplicity of the prompt, we analyzed the errors produced by the test sets and the parts of the source code not covered by the tests, and we tried to improve the prompt to mitigate some problems found. Figure 3 shows the prompt's second version.

Observe that in the prompt presented in Figure 3, `{cut}` and `{code}` have the same meaning, the name of the class under testing and the source code of the class under testing and its dependencies. We were more incisive regarding how we wanted the test set. Including mandatory dependencies, timeout, throws Exception, test set name, and the calling of void methods and default constructors. We also enforce two testing criteria: decision coverage and boundary values.

After this prompt upgrade, we rerun all scripts to generate new test sets, check their quality, and measure coverage and mutation scores. Table 4 presents the average data per temperature. The new prompt improved the test set successful execution by more than 12%, observing temperature 0.2, 64 out of 99 test sets executed without failures, a successful rate of 64.6%. We also improved coverage and mutation scores to an average of 93.5% and 58.8%, respectively.

Table 4: Average Data for Each Temperature Parameter – Prompt version 2

Temp.	# of Suc. Test	% of Suc.	AVG Cov.	AVG Score
0.0	61	61.6	93.5	58.8
0.1	59	59.6	93.4	57.4
0.2	64	64.6	90.7	57.4
0.3	63	63.6	91.2	57.7
0.4	59	59.6	92.0	57.8
0.5	55	55.6	93.3	57.7
0.6	63	63.6	88.0	55.9
0.7	54	54.5	89.9	55.4
0.8	55	55.6	88.6	55.3
0.9	54	54.5	85.8	54.1
1.0	61	61.6	87.7	54.1

Based on this data, we decided to detail the analysis per program and temperature to verify if each temperature has similar behavior for each program. Table 5 presents the data. The first thing we observed in the last two lines of the table is that, in general, the lower the temperature value, the greater the number of programs without successful test sets.

In the worst case, for temperature 0.0, 12 programs out of 33 (36,4%) have no test set running successfully. In the best case, temperature 1.0, 3 out of 33 programs (9.1%) have no test set running successfully. We tried to investigate the reasons, especially for these three programs, why they fail to generate successful runnable test sets. The general observation is that, for these specific programs, they define an `Item` interface and a `MyItem` class implementing the interface, but this class did not override `compareTo` and `equals` methods from `Object` class in Java. Nevertheless, ChatGPT seems to assume they are available for object comparison once several tests use object comparison, but they check reference equality and not object field contents, failing the test cases. This is why all tests for programs 10, 18, and 19 have no test set available, independently of the temperature's parameter.

```
I need functional test cases to cover all
decisions in the methods of the class
under testing.
All conditional expressions must assume
true and false values.
Tests with Boundary Values are also
mandatory. For numeric data, always use
positive and negative values.
All tests must be in one Java class file.
Include all necessary imports.
It is mandatory to throws Exception
in all test method declarations.
It is mandatory to include timeout=1000
in all @Test annotations.
It is mandatory to test for the default
constructor.
Each method in the class under test must
have at least one test case.
Even simple or void methods must have a
test calling it with valid inputs.
@Test(expected= must be used only if the
method under testing explicitly throws
an exception.
Test must be in JUnit 4 framework format.
Test set heather package and import
dependencies:
package ds;
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;
import ds.*;
The class under testing is {clazz}.
The test class must be {cut}Test

Class under testing
*******************

{code}
```

Figure 3: Prompt version 2 for test set generation

Also inspired by Araujo and Vincenzi [3], who observed that by merging test sets from EvoSuite, Palus, JTExpert, and Randoop, the resultant merged test set performs better than any other individual test set in terms of coverage and mutation score, we decided to create a merged test set considering the test sets provided by different temperatures. Moreover, in our case, by merging all test sets, only 3 out of our 33 programs will remain without valid tests. The last column of Table 5 presents the number of valid tests for each program. Only for two programs (9 - Mergesort and 31 - ExactMatch) did we get the maximum number of 33 valid tests, 3 for each different temperature value. Then, we use the JUnit test suite to create a test suite corresponding to all successful test sets. Figure 4 presents an example of a JUnit test suite, considering the 10 successful test sets for program 1 - Max. We built a `ds.All.java` test suite file for each program and used it to collect coverage and mutation scores for all programs. The collected data is shown in Table 6.

```java
package ds;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({ MaxTest2.class, MaxTest5.
    class, MaxTest8.class, MaxTest9.class,
    MaxTest10.class, MaxTest14.class, MaxTest18.
    class, MaxTest20.class, MaxTest22.class,
    MaxTest27.class })
public class All { }
```

Figure 4: Example of JUnit test suite for Max program.

In the two last columns of Table 6, we show the best results [3] obtained considering the merged test set in their experiment. We will refer to our merged test set as LLM Suite and [3]'s merged test set as Baseline Suite. We highlight in gray the cells with the best values concerning the coverage or mutation score of each merged test set.

Regarding code coverage, LLM Suite did not reach Baseline Suite results in 6 out of 33 programs (10, 18, 19, 21, 23, and 26). As already mentioned, for three of these 6 programs (10, 18, and 19), ChatGPT could not create runnable without-fail tests, and we got zero coverage. For all the other programs, both suites covered all program source code. On average, LLM Suite coverage is 90.2%, and Baseline Suite coverage is 99.5%. If we remove programs 19, 18, and 19 from the analysis, Baseline Suite keeps the same coverage of 99.5%, but LLM Suite coverage reaches 99.2%, almost the same.

The biggest surprise occurred with the mutation score. As can be observed, for 14 out of 33 programs, the LLM suite overcome the mutation score of Baseline Suite, and in some cases, it improves by more than 20% the mutation score, like in programs 1, 20, and 30. On the other hand, the baseline suite scores better for 17 out of 33 programs, and we have a tie for two programs, 6 and 7. The average mutation score for Baseline Suite reaches 78.5%, and for LLM Suite, it is 70.5%. Again, removing programs 10, 18, and 19 from our analysis, we got similar mutation scores of 77.6 and 79.5 for LLM and Baseline suites, respectively.

6 DISCUSSION

Our intention with this work starts without expectations. The idea was to investigate the capability of LLM chats, ChatGPT in our experiment, on generating unit test sets, but when we got the first results from these interactions using the very simple prompt presented in Figure 2, we decided to investigate its potential with more emphasis. The final results presented in Table 6 suggest that these prompts have a very good potential, if not to be used as a single way for unit testing generation, its combination in a coordinated way with traditional automatic testing generators can be very promising. Testing will always be a challenging activity, as many useful tools we have to automate this process better.

Table 5: Number of successful tests per temperature per project

ID	Program	Temperature											
		0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	All
1	Max	0	1	0	1	2	0	1	1	1	1	2	10
2	MaxMin1	2	1	2	3	3	2	2	2	0	3	2	22
3	MaxMin2	3	3	3	2	2	2	3	3	2	2	2	27
4	MaxMin3	0	0	0	0	0	0	0	0	0	0	1	1
5	Sort1	3	3	3	3	3	3	3	3	3	3	3	33
6	FibRec	0	0	1	0	1	2	2	2	2	3	2	15
7	FibIte	0	0	0	1	2	1	2	0	3	1	2	12
8	MaxMinRec	3	3	3	3	2	2	3	2	3	2	3	29
9	Mergesort	3	3	3	3	3	3	3	3	3	3	3	33
10	MultMatrixCost	0	0	0	0	0	0	0	0	0	0	0	0
11	ListArray	3	3	3	3	3	2	3	3	2	2	2	29
12	ListAutoRef	3	3	3	2	3	3	3	3	3	2	1	29
13	StackArray	3	3	3	3	3	3	3	3	3	3	2	32
14	StackAutoRef	3	3	3	3	2	3	3	2	3	2	3	30
15	QueueArray	0	0	1	2	0	2	3	3	3	2	2	18
16	QueueAutoRef	3	3	3	3	3	2	3	2	2	3	1	28
17	Sort2	0	0	1	0	0	0	0	0	0	1	1	3
18	HeapSort	0	0	0	0	0	0	0	0	0	0	0	0
19	PartialSorting	0	0	0	0	0	0	0	0	0	0	0	0
20	BinarySearch	3	2	3	2	3	2	2	3	0	0	2	22
21	BinaryTree	3	3	3	3	3	2	3	2	2	3	3	30
22	Hashing1	3	3	3	3	1	2	1	2	2	2	2	24
23	Hashing2	0	1	1	1	0	1	0	0	2	1	1	8
24	GraphMatAdj	3	3	3	3	3	3	3	0	2	2	3	28
25	GraphListAdj1	3	3	2	3	3	1	2	1	1	3	2	24
26	GraphListAdj2	3	3	3	3	3	3	3	2	3	3	2	31
27	DepthFirstSearch	3	2	3	3	2	2	3	2	2	0	2	24
28	BreadthFirstSearch	3	2	3	1	1	1	0	1	1	0	3	16
29	Graph	2	2	2	3	2	2	2	2	1	1	1	20
30	PrimAlg	0	0	0	0	0	0	1	1	0	0	1	3
31	ExactMatch	3	3	3	3	3	3	3	3	3	3	3	33
32	AproximateMatch	3	3	3	3	3	3	3	2	3	2	3	31
33	Identifier	0	0	0	0	0	0	0	1	0	1	1	3
# Successful Test		61	59	64	63	59	55	63	54	55	54	61	648
% Successful Test		61.6	59.6	64.6	63.6	59.6	55.6	63.6	54.5	55.6	54.5	61.6	59.5
# of Programs without test		12	10	8	8	9	8	8	8	9	8	3	3
% of Programs without test		36.4	30.3	24.2	24.2	27.3	24.2	24.2	24.2	27.3	24.2	9.1	9.1

Prompts also show us huge flexibility in asking for test cases considering specific testing criteria or asking for test cases to reach a specific objective, like covering a specific statement or killing a specific mutant. In this work, we decided only on a standard pre-defined prompt, as shown in Figure 3, to use the generated unit testing fully automated, i.e., without interacting with the chat asking for additional testing or testing corrections.

We do not think LLMs will solve all testing problems automatically. We believe a good automated testing strategy now gained important support from LLMs. We intend to observe the LLM limits for unit testing generation. If some important testing requirement is missing, having time and people available for testing, it is possible to develop specialized prompts to solve and generate specific test cases with human support to check and correct possible mistakes. This is especially true once it is difficult to maintain software testing generators. For instance, considering the ones used by Araujo and Vincenzi [3], two of them (Palus and JTExpert) are unavailable or did not work with new versions of Java.

On the other hand, LLMs need a huge amount of data to work and can be easily personalized to meet different testing objectives. A possible alternative to improve the LLM capabilities, considering Java programs, for instance, is to use EvoSuite to start the test set generation and, later, to provide to the LMM the source code of the class under testing and also a previously generated EvoSuite test sets. In this way, we suppose the prompt can better understand the test case style, which may reduce the test case failures generated by LLMs.

Table 6: All LLM test sets versus baseline test sets

ID	LLM Suite		Baseline Suite[11]	
	Coverage	Score	Coverage	Score
1	100.0	85.7	100.0	64.3
2	100.0	85.7	100.0	83.8
3	100.0	85.7	100.0	84.3
4	100.0	64.5	100.0	79.8
5	100.0	80.0	100.0	78.5
6	100.0	100.0	100.0	100.0
7	100.0	100.0	100.0	100.0
8	100.0	83.3	100.0	83.1
9	100.0	96.4	100.0	95.5
10	0.0	0.0	100.0	45.6
11	100.0	93.5	100.0	78.1
12	100.0	87.0	100.0	83.9
13	100.0	96.8	100.0	81.3
14	100.0	93.5	100.0	85.5
15	100.0	93.0	100.0	90.5
16	100.0	67.6	100.0	82.4
17	100.0	57.6	100.0	68.8
18	0.0	0.0	100.0	73.9
19	0.0	0.0	100.0	84.4
20	100.0	94.7	100.0	70.4
21	81.3	62.5	88.8	93.8
22	100.0	57.3	100.0	93.9
23	98.1	63.3	100.0	86.6
24	100.0	73.4	96.2	69.0
25	100.0	78.9	100.0	81.9
26	98.0	77.2	99.2	78.1
27	100.0	78.7	100.0	81.0
28	100.0	78.7	100.0	82.1
29	100.0	78.7	100.0	81.4
30	100.0	71.1	100.0	42.4
31	100.0	39.5	100.0	58.6
32	100.0	38.4	100.0	51.6
33	100.0	64.0	100.0	75.8
AVG	90.2	70.5	99.5	78.5
SD	29.2	27.5	2.0	13.9
AVG**	99.2	77.6	99.5	79.5
SD**	3.4	16.5	2.1	13.1

** - AVG and SD removing zeros.

Another point is that we decided to provide the class under testing source code to LLM and asked it to generate tests for the entire class. However, we believe if you ask for a test only for a specific method inside a class, we will get better results once the scope is reduced, and LLM will create more tests for each specific method.

Finally, we explore a single OpenAI API model called `gpt-3.5-turbo`, but OpenAI offers a variety of models, each with different capabilities. Deciding which one is more suitable for each situation demands additional experimentation. Moreover, there are also a lot of new LLMs available like Bing[12], Bard[13], and LLaMa[14] which may also demand more investigation concerning their capacity on automatic generating unit testing for specific languages.

[12]https://www.bing.com/
[13]https://bard.google.com/
[14]https://labs.perplexity.ai/

7 THREATS TO VALIDITY

There are several potential threats in this paper. One possible threat is sampling bias, which means the selection of programs and tools used in the experiment may not accurately represent the entire software development landscape. This could lead to biased results that may not apply to other contexts. To minimize this threat, we tried to use tools and programs already explored in other experiments. Moreover, especially for the automated test generator, at least EvoSuite is a tool used in a vast number of experiments both in academia [22, 26] and in industry [10] and is also integrated into professionals' integrated development environments [4].

Another threat is the limited generalization of our findings. The study's conclusions may only be relevant to a specific set of programs and tools and may not apply to different scenarios. Additionally, there is a risk of measurement bias, where the metrics used to measure the effectiveness of the generated test data may not fully capture its quality and comprehensiveness. In this way, we manually revise the Python scripts and check the collected data for some programs to ensure the information is accurate. Coverage and mutation scores are traditional metrics for evaluating software testing quality. Specially mutation is confirmed to be an excellent fault model to evaluate the quality of test sets [2, 12, 13]. Although we work with Java programs on this initial investigation, other studies in the course also explore the LLM test generation capabilities for programs written in other languages like Python and C, for instance.

In Section 4.2, we presume the correctness of all programs, as they are basic and known algorithms. However, there remains the possibility of bugs that could result in inaccurate mutation scores and failure to run correct tests.

Using large language models for automatic test data generation may have limitations or biases that could impact the quality and comprehensiveness of the generated test sets. Using baseline results obtained from traditional automated test case generators [3] to confront the results obtained from test sets generated from LLM aims to minimize this threat. Moreover, we only used an LLM engine and model in this experiment, which may not represent the results for other LLMs or models. We intend to extend the experiment for many programs, LLM engines, and models in further studies.

8 CONCLUSION

In this work, we presented an initial investigation of the use of OpenAI API, considering the LLM named `gpt-3.5-turbo`, for unit test generation in a fully automated way, i.e., with no human interaction for test case correction after prompt return. The idea was to detect to which extent the test cases will run directly, with no errors, for testing a set of Java programs.

Basically, we developed a prompt to ask test sets via API, only varying the code of the class under testing and the "temperature" parameter of `gpt-3.5-turbo` model. We asked for three test sets for each one of the eleven different temperature values (0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0) for each program, resulting, in the best case, in a total of 33 test sets per program.

Our results show that not for all temperatures the API was able to produce useful test sets that run automatically with no error without human intervention. In this way, we discarded these test sets in our experiment. For 3 out of 33 programs, the model was not able to generate useful test sets for any temperature, especially due to the non-overriding of traditional Java methods for object comparison like `equals()` and `compareTo()` for the application under testing.

We observed interesting results by keeping only test sets that run automatically and comparing our results with those obtained by other researchers that used traditional automated test set generators [3]. We considered that, besides the simplicity of the prompt, asking for testing to the LLM, the results in terms of code coverage were very similar to the ones obtained in the baseline. Moreover, concerning mutation score, we observed complementary aspects between LLM Suite and Baseline Suite. They complement each other.

Further work intends to investigate the best way to use a traditional automated testing generator and LLM prompts to obtain better results than when using isolated tools.

Moreover, this initial investigation raised more questions than produced answers. To answer the raised questions, more experimentation is necessary. A few of them are:

(1) Do the other OpenAI models produce similar or complementary results?
(2) Does the language used in the prompt influence the results?
(3) Does the language of the product under testing influence the results?
(4) How do other LLMs prompts automate unit testing generation?
(5) Does the LLM perform better by asking testing for a method instead of a class?

REFERENCES

[1] Mehrdad Abdi and Serge Demeyer. 2022. Steps towards zero-touch mutation testing in Pharo. In *21st Belgium-Netherlands Software Evolution Workshop – BENEVOL'2022 (CEUR Workshop Proceedings, Vol. 1)*. Mons, 10.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *XXVII International Conference on Software Engineering – ICSE'05*. ACM Press, St. Louis, MO, USA, 402–411. https://doi.org/10.1145/1062455.1062530

[3] Filipe Santos Araujo and Auri Vincenzi. 2020. How far are we from testing a program in a completely automated way, considering the mutation testing criterion at unit level?. In *Anais do Simpósio Brasileiro de Qualidade de Software (SBQS)*. SBC, 151–159. https://doi.org/10.1145/3439961.3439977

[4] Andrea Arcuri, José Campos, and Gordon Fraser. 2016. Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 401–408. https://doi.org/10.1109/ICST.2016.44

[5] 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. https://doi.org/10.1109/TSE.2014.2372785

[6] Henry Coles. 2015. PITest: real world mutation testing. Disponível em: http://pitest.org/. Acesso em: 04/07/2016. bibtex*[howpublished=Página Web].

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (April 1978), 34–43. https://doi.org/10.1109/C-M.1978.218136

[8] Leo Fernandes, Márcio Ribeiro, Rohit Gheyi, Marcio Delamaro, Márcio Guimarães, and André Santos. 2022. Put Your Hands In The Air! Reducing Manual Effort in Mutation Testing. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering (SBES '22, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 198–207. https://doi.org/10.1145/3555228.3555233 event-place: Virtual Event, Brazil.

[9] 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 (ESEC/FSE '11)*. ACM, Szeged, Hungary, 416–419. https://doi.org/10.1145/2025113.2025179

[10] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (Dec. 2014), 1–42. https://doi.org/10.1145/2685612 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[11] Gordon Fraser and Andrea Arcuri. 2016. EvoSuite at the SBST 2016 Tool Competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, Austin, Texas, 33–36. https://doi.org/10.1145/2897010.2897020

[12] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62 Conference Name: IEEE Transactions on Software Engineering.

[13] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014, Vol. 1)*. Association for Computing Machinery, Hong Kong, China, 654–665. https://doi.org/10.1145/2635868.2635929

[14] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Finding Failure-Inducing Test Cases with ChatGPT.

[15] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. https://doi.org/10.48550/arXiv.2305.09434

[16] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. https://doi.org/10.48550/arXiv.2305.12138

[17] OpenAI. 2023. OpenAI GTP-3.5 Models Documentation. (July 2023). https://platform.openai.com/docs/models/gpt-3-5

[18] 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 (OOPSLA '07)*. ACM, 815–816. https://doi.org/10.1145/1297846.1297902 bibtex*[acmid=1297902;numpages=2] event-place: Montreal, Quebec, Canada.

[19] Marco Tulio Ribeiro. 2023. Testing Language Models (and Prompts) Like We Test Software. (May 2023). https://towardsdatascience.com/testing-large-language-models-like-we-test-software-92745d28a359

[20] M. Roper. 1994. *Software Testing*. McGrall Hill.

[21] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. 2015. JTExpert at the Third Unit Testing Tool Competition. 52–55. https://doi.org/10.1109/SBST.2015.20

[22] Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. 2023. EvoSuite at the SBST 2022 Tool Competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing (SBST '22)*. Association for Computing Machinery, New York, NY, USA, 33–34. https://doi.org/10.1145/3526072.3527526 event-place: Pittsburgh, Pennsylvania.

[23] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. https://doi.org/10.48550/arXiv.2305.00418

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, \Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17, Vol. 1)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010. event-place: Long Beach, California, USA.

[25] Auri M. R. Vincenzi, Tiago Bachiega, Daniel G. de Oliveira, Simone R. S. de Souza, and José C. Maldonado. 2016. The Complementary Aspect of Automatically and Manually Generated Test Case Sets. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2016, Vol. 1)*. ACM, 23–30. https://doi.org/10.1145/2994291.2994295 bibtex*[acmid=2994295;numpages=8] event-place: Seattle, WA, USA.

[26] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. 2021. EvoSuite at the SBST 2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 28–29.

[27] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. https://doi.org/10.48550/arXiv.2303.07839

[28] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. https://doi.org/10.48550/arXiv.2305.04764

[29] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. https://doi.org/10.48550/arXiv.2305.04207

[30] Sai Zhang. 2011. Palus: A Hybrid Automated Test Generation Tool for Java. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. Association for Computing Machinery, New York, NY, USA, 1182–1184. https://doi.org/10.1145/1985793.1986036 event-place: Waikiki, Honolulu, HI, USA.