UNIVERSIDADE FEDERAL DE SÃO CARLOS
PROGRAMA DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Matheus Malonda dos Santos Macaia

Study and Quality Evaluation of LLM's Generated Unit Test Sets for C Programs

São Carlos, São Paulo

2024

Matheus Malonda dos Santos Macaia

Study and Quality Evaluation of LLM's Generated Unit Test Sets for C Programs

Trabalho de Graduação do Curso de Graduação em Engenharia de Computação da Universidade Federal de São Carlos para a obtenção do título de bacharel em Engenharia de Computação.

Orientação Prof. Dr. Auri Marcelo Rizzo Vincenzi

São Carlos, São Paulo

2024

UNIVERSIDADE FEDERAL DE SÃO CARLOS

MATHEUS MALONDA DOS SANTOS MACAIA

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Engenharia de Computação, sendo aprovada em sua forma final pela banca examinadora:

São Carlos, Setembro 2024

Abstract

Context: As technology becomes more integrated into our daily routines, reliable software becomes increasingly critical. However, the high cost of manual test generation often leads developers to neglect software quality concepts. In this context, the growing demand for automated test generation is a crucial response to the potential negative consequences of inadequate software testing. **Problem**: Various tools designed explicitly for automated program testing exist for different programming languages, including C. However, learning and properly configuring these tools is often not trivial, and users must install and set them up for use. **Solution**: This work leverages the rapid rise of Large Language Models (LLMs) to evaluate their capability in generating unit tests for C programs, using code coverage and mutation score as metrics to assess the quality of the generated test sets. **Method**: This study selected 27 C programs from the literature. We grouped these programs into three non-overlapping categories, depending on how each one accepts inputs (Basic Input – inputs provided as program parameters; Driver Type 1 – each test case is a `case` option in a `switch` command and the inputs are hard-coded inside the `case` option; and Driver Type 2 – similar to Driver Type 1 but with the inputs encoded on external data files). For each program, we interactively asked the LLM to generate tests automatically. After generating the test sets, we collected metrics such as code coverage, mutation score, and test execution success rate to evaluate the efficiency and effectiveness of each set. We then used these metrics as new parameters to enhance the efficiency of the sets. **Results**: The test sets generated by LLMs demonstrate significant relevance by presenting substantial results, given the ease of use and low need for human intervention in adjusting the necessary configuration guidelines. On average, LLMs test sets reached 100% of code coverage and 98,7% of mutation score on testing programs with basic inputs. The worst results are in testing programs requiring a driver of Type 1, reaching 91,8% of code coverage and 95.2% of mutation score. Nevertheless, these results are very satisfactory, mainly due to the prompt simplicity and the effort required for test case generation.

Keywords: software testing, automated test generation, coverage testing, mutation testing, large language models, ChatGPT, unit testing

List of Figures

List of Tables

Contents

1 Introduction

Testing software is a complex and time-consuming activity. Still, it is essential to ensure the quality of software products, as well as to enable the approval of software products in CI/CD processes (*Continuous Integration (CI)/Continuous Deployment (CD)*). Such methods, generally adopted in modern software development, are only possible due to the existence of automated tests that aim to ensure that frequent changes do not "break" the software under continuous development.

The field of artificial intelligence (AI) is advancing significantly, mainly due to the recent development and use of LLMs, with the most famous examples being ChatGPT[1] and Bing Copilot[2] which use the GPT-3.5 and GPT-4 models. These models have transformed many fields by demonstrating an extraordinary ability to interpret and produce text with human-like characteristics. Additionally, they can organize and correct files such as documents, code, and even some architectures and generate images and videos.

Given the significant accessibility in interpreting and generating code through high-level language instructions, LLMs show enormous potential in automation in various scenarios, including classical areas of Software Engineering. In the context of software testing, the research area known as SBST (*Search-based Software Testing*) has already been extensively using artificial intelligence techniques to solve testing problems, with the automatic generation of test data being one of the most explored. In this sense, this paper investigates the performance of LLM (ChatGPT 3.5) in generating unit tests for C programs and analyzes the quality of the generated test sets in light of traditional testing criteria, such as code coverage testing and mutation testing.

The results obtained in the study demonstrate the LLMs' capability to generate high-quality tests, considering the programs used in this study. Moreover, prompt interface simplicity is essential in adopting LLM to support different software engineering tasks, including software testing.

The remainder of this paper is organized as follows. In Section 2, the terminology and basic concepts for understanding the work are presented. In Section 3, we describe some related work exploring LLM on testing activities. In Section 4, we describe our experiment design. In Section 5, we present the collected data and the analysis we performed. In Section 6, we describe the lessons learned on using LLM for testing C programs. Finally, in Section 7, we conclude the paper and point out some future work.

[1] <https://chatgpt.com/>
[2] <https://www.bing.com/copilot>

2 Background

2.1 Software testing

Testing software is a fundamental activity in the software development cycle. Its main objective is to demonstrate the presence of faults in the product under test.

Software testing is categorized into 4 stages: (i) planning, (ii) test case design, (iii) execution, and (iv) analysis. Once, in general, programs have a very large or infinite input domain, test case design is of fundamental importance once, as stated by Roper (1994), testing is sampling. The quality of the test activity depends on a good test sampling.

The testing phases are defined based on the focus, needs, and problems to be addressed. Thus, different types of tests are applied at different stages, with the most common being:

Unit testing: aims to verify the proper functioning of individual software components, such as functions, methods, or classes;

Integration testing: examines how various software modules or components interact. It ensures that individual units exchange data appropriately and integrate as planned;

System testing: evaluates the system as a whole. It involves verifying that all parts and functionalities work as intended by testing an entire application that mimics the production environment.

Acceptance testing: which aims to verify whether a system meets the customer's or end-user's needs and expectations. In a production or pre-production environment, the customer or a customer representative usually conducts these tests.

Any software product can be tested primarily through functional and structural testing. In functional testing, the program is evaluated solely based on its specifications; the tester does not have access to the internal structures or implementation of the software. Structural testing, on the other hand, involves generating test cases based on the program's internal structure. In addition to functional and structural testing methodologies, fault-based testing involves using knowledge of common faults committed during the software development process.

Each testing technique has a set of testing criteria which, based on a source of information, like a program specification or a program source code, derives test requirements that a test set must obey to become adequate with respect to (w.r.t.) that specific testing criterion. Our study emphasizes test sets targeting code elements, so we concentrate on trying to generate adequate test sets for structural and fault-based testing criteria.

This study focuses on generating unit test cases with support from large language models (LLM). Moreover, to judge the quality of the generated test set, we will use traditional software testing criteria, namely Statement Coverage (ROPER, 1994) and Mutation Testing (DEMILLO; LIPTON; SAYWARD, 1978).

Once software testing is generally neglected, our intention is to provide automated support for test case generation, reducing the effort of manual testing. Manual testing, while effective, tends to be more expensive and subject to errors committed by the tester during the test case creation and automation.

Mutation testing is considered an excellent fault model for judging the quality of test sets (ANDREWS; BRIAND; LABICHE, 2005; PAPADAKIS et al., 2018). It involves adding intentional faults within the program to verify if the test set can detect these faults and return something different from expected or if it fails to trace them, generating the same output as the original program. Once each mutant represents a possible fault, the idea behind mutation testing is to find a test set able to expose the difference between each mutant and the original program.

When this happens, we can ensure the original program did not contain the fault modeled by the mutant.

Mutants are generated with the aid of mutation operators, defined according to the characteristics of the programming language in which the programs are being implemented. Not all mutants can be considered a fault. Some syntactic changes may produce an equivalent implementation of the original program, i.e., it may create an equivalent mutant. As the determination of equivalence is an undecidable problem, in general, the task of determining equivalence is performed manually.

To measure the adequacy of a given test set T, considering the mutation testing applied in a program P, we compute the mutation score MS. MS is the ratio between all dead mutants of P killed by T and the total number of non-equivalent mutants of P. This ratio varies from 0 (0%-adequate) to 1 (100%-adequate), so closer to one means a more adequate test set.

$$MS(T,P) = \frac{DM(P,T)}{TM(P) - EM(P)}$$

Where:

- **MS(P,T):**	Mutation score of T on testing P.
- **DM(P,T):**	Dead (killed) mutants of P by T.
- **TM(P):**	Total number of generated mutants.
- **EM(P):**	Number of equivalent mutants.

When MS is equal to 100%, the test set T is adequate for testing the program P w.r.t. mutation testing.

Both the automatic test case generation to kill mutants and the determination of equivalent mutants are, in general, undecidable problems and very costly tasks when performed manually. In this sense, although there are several limitations, artificial intelligence (AI) techniques are extensively used to partially help solve both problems. The area known as search-based software testing (SBST) employs AI to find approximate solutions to undecidable software testing problems.

Search-based software testing (SBST) helps us with several testing tasks (COHEN, 2017), such as automated test data generation (FRASER; ARCURI, 2011; ROJAS et al., 2017; JATANA; SURI, 2020), automatic identification of infeasible test paths (DING; TAN; LIU, 2012; DING; TAN, 2013; NETO et al., 2022), automatic identification of equivalent mutants (SILVA; SOUZA; SOUZA, 2017; KUSHARKI et al., 2022), test case prioritization (SINGH et al., 2023), and so on.

2.2 Automatic test data generation

There are several traditional approaches to automatic test data generation, including methods based on static and dynamic analysis, as well as search-based and machine learning techniques (TAHBILDAR; BORBORA; G.P., 2013).

It is not trivial to use tools for automatic test generation. These tools often require complex configurations and a deep understanding of the software under test. Since a program can have many inputs, it is important to efficiently choose which inputs to use in testing. The number of possible inputs, in general, is very large. The generators must infer a type for test data generation in a dynamically typed language. Moreover, we need both the test input and the expected output to create a test case. Generating the correct expected output for a given input is known as an oracle problem (BARR et al., 2015).

Traditional automated test case generation, such as EvoSuite (FRASER; ARCURI, 2016) and Randoop (PACHECO; ERNST, 2007) for Java or Pynguin (LUKASCZYK; FRASER, 2022) for Python, generate the so-called “regression test cases” (PACHECO; ERNST, 2007), i.e., all generated test cases pass in the existing product implementation once they assume the current output as the expected output. So, generated test sets are useful to validate future changes in the current implementation, avoiding the oracle problem.

New AI advances with the advent of the Large Language Models (LLM) are making access to AI tools more popular. LLM's impact on the workforce of several professions is still unclear (ELOUNDOU et al., 2023). Specifically, on software testing tasks, we expect a positive impact on productivity and time-saving by using LLMs. Therefore, it is natural to evaluate how LLM can help in different activities, including software engineering (OZKAYA, 2023; RASNAYAKA et al., 2024) and testing activities (NGUYEN et al., 2023; PIYA; SULLIVAN, 2024).

2.3 Large Language Models – LLMs

LLMs are a new class of machine learning models (GeeksforGeeks, 2024), being a type of artificial intelligence algorithm that uses self-supervised learning techniques to process and understand text or human languages using neural network techniques with a large number of parameters. Their performance and aptitude in a variety of tasks, including chat, open question answering, content summarizing, execution of almost arbitrary instructions, translation, and content and code generation, have far surpassed the performance of their predecessors like N-gram models (JURAFSKY; MARTIN, 2009), SHRDLU (WINOGRAD, 1971), and ELIZA (WEIZENBAUM, 1966).

Several aspects influence the architecture of LLMs, such as the specific model design purpose, available computational resources, and the types of language processing tasks the LLM is expected to perform. Multiple layers, including feed-forward layers, embedding layers, and attention layers, make up the overall architecture of an LLM, as depicted in Figure 1.

Predictions are produced by working in conjunction with embedded text. The Transformer architecture is the main foundation of modern LLMs and can be characterized by the following components:

Encoder: This component transforms large volumes of text into tokens – numerical values. Words with comparable meanings are grouped in the vector space by the Encoder, which generates meaningful token embedding;

Embedding: Transforms tokens into high-dimensional vectors, with positional information added to the embedding so that the word order is considered;

Attention Mechanism: Allows LLMs to effectively handle large amounts of information by observing the relationships between all tokens in a sequence. Based on the calculation of attention scores, this method allocates different levels of importance to various tokens, enabling the establishment of dependencies and relationships based on context. As such, it allows models to focus on specific segments of the input text, particularly correlated words or phrases;

Feed Forward: Its high parallelization makes efficient processing of large sequences possible, where the feed-forward applies a neural network separately to each token, performing non-linear transformations to the token embeddings, which helps capture intricate relationships and complex non-linear patterns in the data;

Decoder: The decoder uses the encoder's internal representation to create the output sequence. Each block is repeated n times, and at each step, it selects the word with the highest probability, generating an output sequence where tokens are passed through a linear layer, transforming them back into text.

Evaluating LLMs is a challenging and constantly evolving field, mainly because LLMs often demonstrate varying levels of skill across various tasks. With this in mind, this article focuses on analyzing their efficiency and effectiveness in generating unit test sets and the quality of these generated test sets. Moreover, we discuss the simplicity of interaction with an LLM compared to automated test generators. In our opinion, prompt simplicity is the biggest revolution LLM has caused in supporting different tasks with a simple text interface.

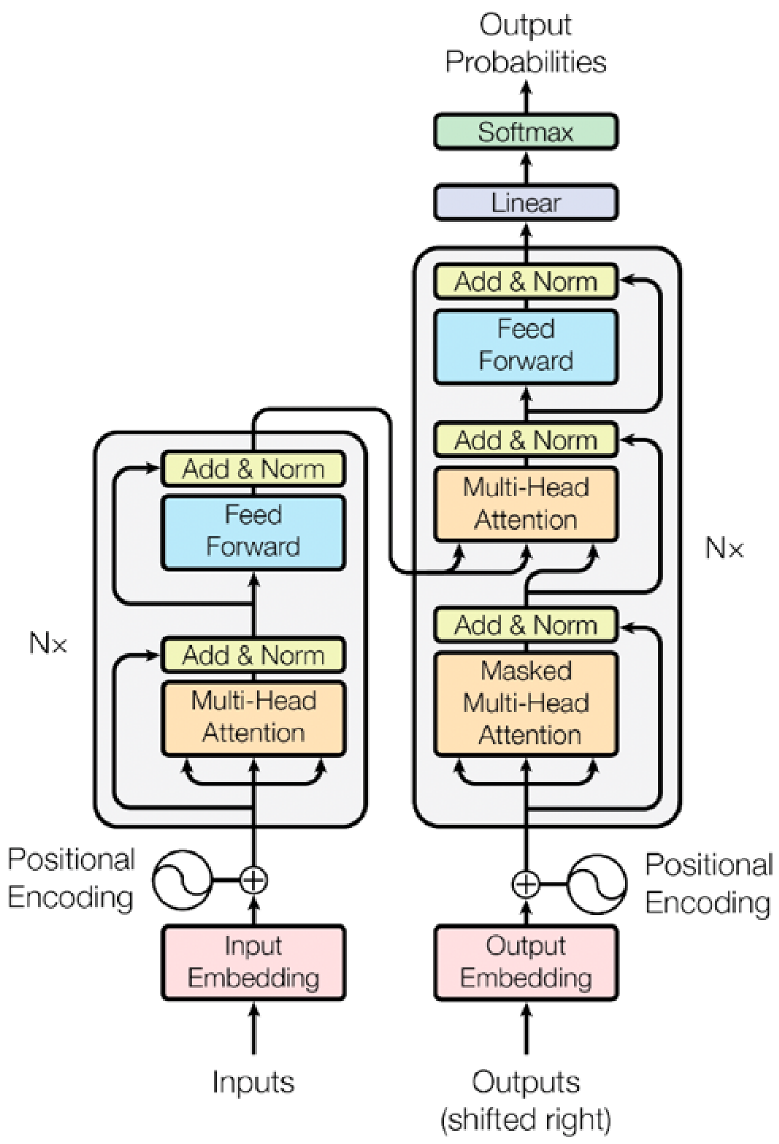

Figure 1 – Components of a Transformer Architecture. (extracted from AWS (2024))

3 Related Work

LLMs, like ChatGPT[1], are state-of-the-art language models based on the Transformer architecture (VASWANI et al., 2017). They are designed to process human language, enabling machines to generate coherent and contextually relevant text. These models are trained on large amounts of linguistic 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. (2023) conduct a comprehensive exploration of the applicability of ChatGPT and its potential in software engineering. The authors examine various tasks, including code generation, code summarization, bug detection, and code completion, to evaluate the performance of ChatGPT. Through 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 ChatGPT's capabilities and offer guidance on harnessing its potential to improve software development practices while highlighting areas where further advancements are needed.

White et al. (2023) also explores the potential applications of ChatGPT in various software engineering tasks. The researchers introduce prompt patterns to leverage ChatGPT's language generation capabilities to enhance code quality, refactoring, requirements elicitation, and software design tasks. Through experiments and case studies, they demonstrate the effectiveness of using ChatGPT with these prompt patterns to assist developers and software engineers in their daily activities. The paper highlights the versatility of ChatGPT as a tool to support software engineering practices and promote better code development and design.

Exploratory studies also investigate using LLMs to generate test data at different stages of testing, from unit tests to end-to-end tests. Notably, the context provided to the LLM was the only aspect that changed during these experiments. In the case of unit tests, the LLM received code snippets as input (LI et al., 2023; YUAN et al., 2023; SIDDIQ et al., 2023; XIE et al., 2023; GUILHERME; VINCENZI, 2023; TANG et al., 2023). For example, a prompt could be formulated as follows:

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

The LLM then used its language generation capabilities to produce comprehensive

[1] <https://chat.openai.com/>

test data sets that addressed various test scenarios. From the studies above, the majority used LLM in an interactive way, such that the results are obtained after a “few-shot” prompting to the LLM (YONG et al., 2023; HURLEY; OKYERE-BADOO, 2024). On the other hand, in some other studies, like (GUILHERME; VINCENZI, 2023), the authors used a “zero-shot” prompt (YONG et al., 2023; HURLEY; OKYERE-BADOO, 2024) strategy to generate test sets without human intervention.

On the other hand, for end-to-end tests, the LLM was provided with a description of the system’s functional specifications (RIBEIRO, 2023) or a GUI (LIU et al., 2023). The prompt may have asked the LLM to:

“Generate test cases that validate all system 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, simplifying testing efforts, and enhancing software quality. By adjusting the input context to the capabilities of the LLM, effective test cases were obtained for different testing stages, further demonstrating the versatility and adaptability of LLMs in software testing. Specifically for test generation, another advantage is that by using LLM, we do not need to worry about specific testing tools for each programming language we use. Generally, it is expected that the LLM can generate test cases for each specific programming language with minor prompt changes.

In the experiment described below, we intend to evaluate the quality of test sets generated by LLM for C programs, considering different program categories related to how the programs received their inputs. We adopt a few-shot prompt strategy and interact with the LLM, trying to maximize code coverage and mutation score metrics. The experiment design, data collection, and analysis are presented in the following sections.

4 Experimental Design

The aim of this study is to evaluate the effectiveness of large language models (LLMs) in creating test data sets for programs developed in the C language. ChatGPT, model version 3.5, was chosen as the main tool for this study due to its accessibility, ease of use, active community, extensive documentation, popularity, and low cost if used via API or free in case of use of the interactive prompt (our choice).

These features make ChatGPT one of the most user-friendly and accessible tools, making it an excellent candidate for this study. However, in future works, we intend to explore other LLMs and compare them when carrying out the same task.

We used statement coverage and mutation testing to evaluate the efficiency and effectiveness of the LLM's generated test sets, as previous research (DELAMARO et al., 2021) (DELAMARO et al., 2014) has demonstrated that this method is robust and effective for experimentation and assessment of test set quality.

Mutation testing follows the processes outlined in Section 2. ChatGPT, the study's target tool, will interactively receive instructions and requests based on specific code needs, using a few-shot prompt strategy

Once we receive a test set, we try to compile and run it in the original program and generate mutants, also measuring the statement coverage of the generated test set.

For this study, 27 programs written in C were chosen, previously used in experimental studies by other researchers (DELAMARO et al., 2014; DELAMARO; OFFUTT; AMMANN, 2014; AMMANN; DELAMARO; OFFUTT, 2014; ANDRADE et al., 2019; DELAMARO et al., 2021) to ensure the relevance and comparability of the results.

Table 1 lists the selected programs, the number of mutated functions (#Fun) on each program, the number of lines to quantify the program size in terms of lines of code (LOC), helping to define the complexity of the programs, the number of generated mutants (#Mut) considering all unit mutation operators implemented in the Proteum/IM (DELAMARO; MALDONADO; VINCENZI, 2001) mutation testing tool for C, representing all the faults a test set must identify and the number of equivalent mutants (#Eq). Observe that in the work of Delamaro et al. (2014), they created an adequate test set and determined the equivalent mutants for all these 27 programs manually. We simply used these previously generated data such that, in our experiment, we only used the non-equivalent mutants to evaluate the quality of the LLM-generated test cases.

From Table 1, we can observe that all these programs have 78 functions, implemented on 1374 lines of code and, considering all unit mutation operators, Proteum/IM generated

23,253 mutants, being 1,864 (8,0%) equivalents on average. To kill all the non-equivalent mutants Delamaro et al. (2014) created 1086 test cases manually, presented in Table 5 on pp. 33.

Table 1 – Information Table of Programs (data adapted from Delamaro et al. (2014)

Program	**#Fun**	**LOC**	**#Mut**	**#Eq**
boundedQueue	6	49	1121	100
cal	1	18	891	71
Calculation	7	46	1118	107
checkIt	1	9	104	5
CheckPalindrome	1	9	166	21
countPositive	1	10	151	9
date-plus	1	132	2421	164
DigitReverser	1	9	496	45
findLast	1	17	198	8
findVal	1	16	190	14
Heap	7	43	1079	104
inversePermutation	7	41	576	61
jday+date	1	45	2821	84
lastZero	1	9	173	10
LRS	2	54	1132	290
MergeSort	1	126	991	32
numZero	1	9	151	12
oddOrPos	1	10	361	71
power	8	18	268	8
printPrimes	2	35	715	78
Queue	6	233	469	38
quickSort	1	60	1026	66
RecursiveSelecSort	2	101	555	37
Stack	5	56	461	53
sum	7	45	165	18
trashAndTakeOut	1	55	599	19
UnixCal	4	119	4855	339
Sum	78	1374	23253	1864
Average	2.89	50.89	861.22	69.04
Standard Deviation	2.59	51.82	1033.14	81.16

ChatGPT's task is to receive high-level text instructions close to human interaction and generate a test data set for each program under test. Then, the quality of the generated test set, in terms of its efficiency in executing the code under testing and effectiveness in detecting defects, is analyzed by computing the statement coverage and the mutation score of the set against all non-equivalent unit mutants generated by Proteum/IM (DELAMARO; MALDONADO; VINCENZI, 2001).

We need to compile the original program before using Proteum/IM for mutant generation. Each program has a `compile.txt` file containing the correct `gcc` parameters for program compilation. We also generated an instrumented version of the original program to compute statement coverage using `gcc` with the `-coverage` parameter.

Then, we created a Proteum/IM test session for each program and used all unit mutation operators available to generate all possible mutants, discarding the equivalent ones previously determined by other researchers (DELAMARO et al., 2014) (see Table 1).

Thus, after generating all mutants and discarding the equivalent ones, the experimental cycle for each program consists of:

1. Request a test set via LLM prompt;
2. Execute the test set against the original program registering the output for each test case;
3. Execute the test set against the instrumented version of the original program, generating the statement coverage report;
4. Execute the test set against all non-equivalent mutants;
5. Calculate the mutation score.
6. If the score is not 100% or if there are uncovered lines. If it's not the final interaction, feedback prompt is given to the LLM, and the cycle restarts

We iterate manually over steps 1 to 5 until it reaches 100% of statement coverage and mutation score or until LLM does not improve coverage and mutation score after 5 interactions, whenever an improvement was detected, the counter for the 5-interaction limit was reset. Figure 2 illustrates the interaction with the LLM aiming to generate good test sets.

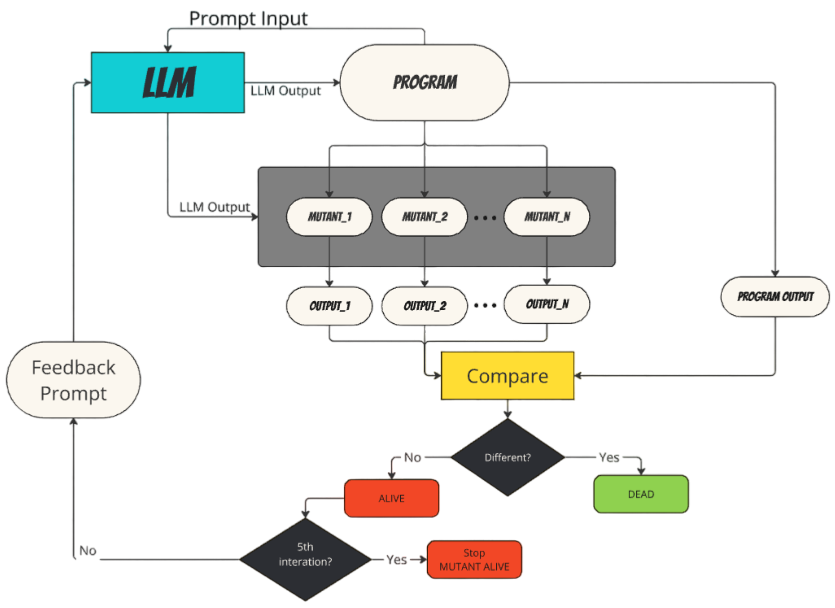

Figure 2 – LLM cycle to kill mutants

5 Data Collection and Analysis

Following the first stage of mutation testing, we need to generate the mutants of the programs. Although we can do this activity manually, it is time-consuming and subject to human mistakes. We used all mutation operators for unit testing implemented in the Proteum/IM testing tool for this task. Proteum/IM can be executed via scripts or GUI interface. We developed a set of scripts to run Proteum/IM via scripts. The complete repository of our experiment is available at <https://anonymous.4open.science/r/c_sbqs2024-0A03/>.

Scripts can compile all programs, generate mutants, execute the original program and all mutants with the generated test set, and produce coverage and mutation testing reports. This tool saves considerable study time by automating the generation and validation of mutants. Since the programs have already been used in previous research, their equivalent mutants are also mapped and discarded from our analysis, so we use only non-equivalent mutants.

By using LLM to generate a test set, we adopted the following strategy: requests must be made clearly and specifically, leaving no room for ambiguity and ensuring that the responses are precise and useful. Since LLMs work by storing information passed to them in tokens, which have a cost associated with the amount used, and considering that the search should focus on generating better tests with the fewest possible resources, it is not beneficial to spend resources on many maintenance requests.

To increase efficiency, ChatGPT will have full access to the code of the analyzed programs, allowing for a thorough and detailed "understanding". This method reduces the need for a lengthy explanation of the type and format of the desired output. One of the guidelines is to analyze the code as an experienced programmer whose goal is to generate a high-quality test set capable of discovering and achieving high statement coverage and mutation score, thereby ensuring better quality in its responses.

To make the test scenarios as standardized as possible, a preliminary analysis of the programs was conducted to generate a common question for all of them. It was noted that they could be categorized into two groups: one in which the program's input variables were already passed as parameters in its call and another in which the inputs were obtained through an external file named `drive.c`. Thus, specific forms of requests to ChatGPT were defined following these categories.

5.1 Basics

These programs receive inputs directly in their call, not requiring any external files to receive inputs. For these, it is only necessary to provide the program to ChatGPT and request that an efficient set of inputs be generated to eliminate all mutants, using the best techniques available in the field of software quality, as shown in the prompt presented in Figure 3. The tag `<<source-code>>` (line 10) is replaced by the complete source code of the original program under testing.

```
1  It is necessary to test this program, aiming to
2  cover all its statements and kill all possible
3  mutants. So, you have to create a set of test
4  cases for this program. The suite needs to be
5  robust enough to cover multiple scenarios, and
6  as an experienced programmer, you should use
7  efficient software quality testing techniques
8  for greater reliability in the test suite.
9
10 <<source-code>>
```

Figure 3 – Example of LLM prompt to deal with programs of Category 1

Each test data for programs of this category is encoded as a line in a text file called `textset.txt`. Each line in this file corresponds to input parameters used to execute the original program and its mutants. Figure 4 illustrates the manual test set for the `cal` program, which calculates the number of days between two dates in the same year.

```
1 1 1 1 1963
2 1 8 24 1963
2 1 8 24 1972
2 1 8 24 1900
2 1 8 24 2000
8 2 8 24 1972
8 24 9 24 1963
7 24 9 24 1963
```

Figure 4 – Example of `testset.txt` file for a specific program of Category 1

5.2 Multiple

These programs use an external file named `driver.c` to process the inputs of the main program. In this case, we need to provide more information to ChatGPT, as poorly defined inputs can lead to unsatisfactory responses.

ChatGPT needs to “understand” the functioning of multiple files, always considering the token limitation. This limitation can lead to the loss of relevant information when correcting unwanted responses and detailing the request needed to obtain the desired output. To circumvent this problem, we need to provide examples of the desired output formats. Additionally, there are two different types of drivers:

Driver Type 1

Programs with driver Type 1 are those where an additional file named `driver.c` contains specific function calls where the order and value of input parameters matter. `driver.c` encodes each test as `case` option inside a `switch` statement. In this scenario, the `testset.txt` file, which previously was responsible for receiving inputs to test the programs, plays a different role, essentially storing the test indices to be executed from the `driver.c` file. Therefore, it is necessary for ChatGPT to understand not only what the program does but also how the `driver.c` is configured, to provide inputs that are efficiently accepted by the program.

For this purpose, part of the content of the `driver.c` file was provided along with the program to generate the input sets, as shown in Figure 5.

```
I need to test each function of the C code below:

<<source-code>>

Test cases are decoded as a driver, as in the
example below:

<<test case encoding sample>>

Generate test cases for each function to kill
as many mutants as possible.

Each test, mapped in a case inside the switch,
must call as many functions as possible and at
least one of such functions must print some output
so that we may use it to show the difference
between the original program and the mutated
program outputs.

Provide just the C code corresponding to the main
function, the switch case, and the test cases. Do
not include additional information before or after
the code. I need just the code.
```

Figure 5 – Example of LLM prompt to deal with programs of Category 2

For example, Figure 6 and Figure 7 illustrate the content of `testset.txt` and `driver.c` for the `boundedQueue` program. This program implements a bounded queue in C, based on an example from the book of Ammann e Offutt (2008).

```
1
2
...
```

Figure 6 – Example of `testset.txt` file for boundedQueue program of Driver Type 1

```c
#include <stdio.h>
#include <malloc.h>
#include "boundedQueue.h"
void driver(int argc, char  *argv[]) {
   int tc_number;
   BoundedQueue  * q, * p;
   tc_number = atoi(argv[1]);
   switch (tc_number) {
        case 1:
            q = createQueue(5);
            enqueue(q,1);
            enqueue(q,2);
            enqueue(q,3);
            enqueue(q,4);
            printBoundedQueue(q);
            free(q);
            break;
        case 2:
            q = createQueue(0);
            free(q);
            q = createQueue(-1);
            printf("%d\n", q == NULL);
            free(q);
            break;
        ...
    }
}
```

Figure 7 – Example of the `driver.c` file for the boundedQueue

Driver Type 2

Type 2 driver programs are similar to Type 1, where test inputs are provided through a third-party file. However, the driver behaves as a decoder for the inputs, so the aforementioned `testset.txt` file once again contains the inputs to test the program. Therefore, in addition to understanding what the program needs, ChatGPT also needs to know the format of the inputs `driver.c` file requires to read and direct the inputs to the program. Thus, the program, examples of the expected input format, and the `driver.c` file will be provided, as shown in the prompt in Figure 8.

For instance, considering the program `MergeSort`, the command line calendar program of Unix and Linux systems, Figure 9 illustrates an example of the content of

```
I need to test each function of the C code below:

<<source-code>>

Test cases are decoded as a driver, as in the
example below:

<<test case encoding sample>>

And here's the example file containing the
inputs for the example tests of the C code:

<<test set source-code>>

Once the execution begins in the main function,
I need test inputs, as in the example format
above, to allow me to reach each additional
function presented in this code. The test input
is used to kill as many mutants as possible.
```

Figure 8 – Example of LLM prompt to deal with programs of Category 3

`testset.txt` file with valid and inputs for this particular program.

```

abc
abc 3
1752
13 1963
```

Figure 9 – Example of `testset.txt` file for MergeSort program – Driver Type 2

Figure 10 shows the code of the `driver.c` program, and Figure 11 shows a piece of `MergeSort.c` programs with the main function delegating to the `driver.c` the input processing.

```c
#include <stdio.h>

void driver(int tc_number, int argc, char  *argv[]) {
    switch (tc_number) {
        case 0:
            dispatch(argc, argv);
            break;
    }
}
```

Figure 10 – Example of the `driver.c` file for the MergeSort

Once we obtained a test set, Proteum/IM ran it against the original program, recording the output of each test input. Observe that we are assuming all subject programs

```c
#include <sys/types.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
...
main(argc, argv)
int   argc;
char  *argv[];
{
    if ( argc > 1 && strcmp("-", argv[1]) == 0 ) {
        driver(atoi(argv[2]), argc, argv);
    }
    else {
        driver(0, argc, argv);
    }
    return 0;
}

dispatch(argc, argv)
char *argv[];
{
    ...
}
...
```

Figure 11 – Piece of `MergeSort.c` program

are correct, and in this way, the output they provide for each test input is assumed to be the correct output, avoiding the oracle problem (BARR et al., 2015).

We also ran all tests with the original program instrumented version, which was previously generated, and after the execution, we generated the coverage report with `lcov` tool[1].

Finally, Proteum/IM ran the test set against all mutants for a specific program and generated a report containing the following information: the total generated mutants, the number of equivalent mutants, the number of live mutants and the current mutation score.

If the program presented statement coverage or mutation score below 100%, we analyzed the test set and provided specific feedback to search for possible problems. If corrections are needed, we request them. Additionally, we interacted with ChatGPT, asking them to correct and/or improve the generated test set and keep the existing ones.

For each new test set, scripts are rerun to recollect data. After five interactions without increment in statement coverage and mutation score, we stopped and followed the next program. The next section presents the results we obtained with this process.

[1] <https://github.com/linux-test-project/lcov>

5.3 Results

The initial goal of this work was to investigate the capability of LLM, specifically ChatGPT, to automatically generate unit tests for C programs. However, upon testing some programs using very simple prompts, where only the format and expected output for the programs were explained, it was noted that the results were satisfactory, especially in simpler programs where no very complex treatment was needed. We need additional work for programs with more complicated output, such as those with drivers Type 1 and 2 mentioned above, but we also consider the results satisfactory.

Tables 2 to 4 presents the results obtained for all 27 subject programs. They are grouped by their input category. Table 2 presents the 19 programs with basic input.

Table 2 – Manual Tests vs. Automatic (LLM) Tests – Basic

Program	**Manual Tests**		**LLM Tests**	
	Coverage	**Score**	**Coverage**	**Score**
cal	100.0	100.0	100.0	94.6
checkIt	100.0	100.0	100.0	99.0
CheckPalindrome	100.0	100.0	100.0	95.9
countPositive	100.0	100.0	100.0	99.3
DigitReverser	100.0	100.0	100.0	100.0
findLast	100.0	100.0	100.0	100.0
findVal	100.0	100.0	100.0	99.4
Heap	100.0	100.0	100.0	98.7
InversePermutation	100.0	100.0	100.0	98.1
jday-jdate	100.0	100.0	100.0	97.0
lastZero	100.0	100.0	100.0	100.0
LRS	100.0	100.0	100.0	97.9
numZero	100.0	100.0	100.0	100.0
oddOrPos	100.0	100.0	100.0	96.6
power	100.0	100.0	100.0	100.0
printPrimes	100.0	100.0	100.0	100.0
quicksort	100.0	100.0	100.0	99.5
RecursiveSelectionSort	100.0	100.0	100.0	100.0
sum	100.0	100.0	100.0	100.0
Average	100.0	100.0	100.0	98.7
Standard Deviation	0.0	0.0	0.0	1.6

We decided to keep in the table the Coverage and Mutation Score determined by the manually generated test sets taken from Delamaro et al. (2014) work. This shows that for each program, it is possible to cover all statements and kill all non-equivalent mutants with the cost of manually generating test cases and determining equivalent mutants. Considering this group of programs, the LLM's generated test set also obtained 100% of statement coverage for all programs and 100% of mutation score for 8 out of 19 programs. The final average mutation score reached 98.7%, and only 3 out of 19 programs had a mutation score below 97.0%.

Table 3 presents the four programs which require a driver Type 1. For these groups of programs, manual testing determined 100% of statement coverage and mutation score. The LLM's generated test set reached 100% of statement coverage on 3 out of 4 programs. Except for Stack, all other programs also had mutation scores above 94.9%. Stack reached 91.8% of statement coverage and 93.6 of mutation score. On average, statement coverage and mutation scores were 98.0% and 95.2%, respectively.

Table 3 – Manual Tests vs. Automatic (LLM) Tests – Driver Type 1

Program	**Manual Tests**		**LLM Tests**	
	Coverage	**Score**	**Coverage**	**Score**
boundedQueue	100.0	100.0	100.0	94.7
Calculation	100.0	100.0	100.0	97.5
Queue	100.0	100.0	100.0	94.9
Stack	100.0	100.0	91.8	93.6
Average	100.0	100.0	98.0	95.2
Standard Deviation	0.0	0.0	4.1	1.7

Table 4 presents the four programs which require a driver Type 2. Again, manual testing determined 100% of statement coverage and mutation score for these program groups. The LLM's generated test set reached 100% of statement coverage on 3 out of 4 programs. `data-plus` was the program with the lowest coverage (99.1%) and mutation score (95.5%). On average, statement coverage and mutation scores were 99.8% and 95.1%, respectively.

Table 4 – Manual Tests vs. Automatic (LLM) Tests – Driver Type 2

Program	**Manual Tests**		**LLM Tests**	
	Coverage	**Score**	**Coverage**	**Score**
date-plus	100.0	100.0	99.1	95.5
MergeSort	100.0	100.0	100.0	97.0
trashAndTakeOut	100.0	100.0	100.0	91.0
UnixCal	100.0	100.0	100.0	97.0
Average	100.0	100.0	99.8	95.1
Standard Deviation	0.0	0.0	0.5	2.8

Finally, Table 5 compares the number of generated test cases using LLM and the number of manual tests. It is important to consider here that we did not apply minimization strategies or eliminate redundant LLM-generated test cases. In the same way, we cannot guarantee there are no redundant test cases from the manual tests taken from (DELAMARO et al., 2014).

It is notable that the size of tests generated by LLM can appear arbitrary if not enough information is provided. Without clear direction, the model may apply different tactics, generating tests with similar objectives but which may not cover all parts of the program. This can result in a large number of ineffective or redundant tests. Therefore, unless you are looking to create extremely specific tests, something that is not as practical due to the limit of information that the model can process, it is more efficient to ask the LLM to create specific tests for particular lines of code or functions.

When tests fail to eliminate mutants, this fine-grained approach may be more effective. On average, LLM generated more than twice (111.1%) as many test cases as manual testing but with a minimum effort in terms of time spent understanding the source code for test case generation. In only two programs, `date-plus` and `Stack`, the number of LLM tests was inferior to manual testing. In the case of `date-plus`, the generated test set reached 99.1% of statement coverage and 95.5% of mutation score (see Table 3). In the case of Stack, we got 91.8% statement coverage and a mutation score of 93.6%.

Table 5 – Comparison of the number of tests generated by each type of test

Program	**#Manual Tests**	**#LLM Tests**	**% Inc.**
boundedQueue	16	18	12.5
cal	40	70	75.0
Calculation	13	16	23.1
checkIt	27	47	74.1
CheckPalindrome	8	14	75.0
countPositive	15	132	780.0
date-plus	313	219	-30.0
DigitReverser	5	43	760.0
findLast	24	111	362.5
findVal	25	116	364.0
Heap	40	49	22.5
InversePermutation	43	200	365.1
jday-jdate	55	93	69.1
lastZero	12	26	116.7
LRS	10	35	250.0
MergeSort	165	575	248.5
numZero	11	29	163.6
oddOrPos	30	31	3.3
power	18	20	11.1
printPrimes	7	9	28.6
Queue	12	18	50.0
quicksort	73	75	2.7
RecursiveSelectionSort	33	74	124.2
Stack	11	9	-18.2
sum	8	73	812.5
trashAndTakeOut	30	80	166.7
UnixCal	42	111	164.3
Sum	1086	2293	111.1
Average	40.2	84.9	188.0
Standard Deviation	63.0	112.2	244.2

We consider the obtained results very satisfactory. The prompts had a significant impact and great potential for test case generation considering programs demanding different types of inputs. The simplicity of handling and high flexibility in the calls make LLM a very relevant tool not only for creating test sets but also for quick expansion, replication, and formatting of code.

This study, therefore, assesses whether LLMs using mutation testing concepts, having behavior and peculiarities already explored in other researchers' work (GUILHERME; VINCENZI, 2023), are also efficient in generating test data for C programs, using a well-known effective analysis method and a well-practiced set of programs. The obtained results contribute to a better understanding of the potential of LLMs to enable the development of high-quality unit test sets, reducing human effort and cost.

6 Lessons Learned

Several points are worth mentioning throughout this research due to their relevance to its development, especially regarding the exploration of ChatGPT's understanding as a tool for generating input sets for programs.

Firstly, it is interesting to note that to determine the most efficient method of instructing it to return the necessary responses to generate an efficient data set, we practice with a sample program asking for test cases using different prompts for ChatGPT. It was observed that to get an efficient response, it is necessary to provide as much information as possible, be as objective as possible, and avoid ambiguities so that there are no unnecessary tokens or accidental misdirection of its focus. It was also noted that it is more efficient to demonstrate than to explain, so it is better to provide, if possible, the basis of what is being requested via prompt, whether text, files, or images.

Depending on the complexity of what is being requested, it is unlikely that a completely correct response will be obtained with a zero-shot prompt. Therefore, it is extremely valuable to give continuous feedback until the desired output is achieved. However, as mentioned earlier, there is a cost associated with using some LLMs, and there is a limit to the amount of information they can handle (context window). If a considerable amount of feedback is given, part of the previous context may be lost, and in this process, relevant information for the desired output might be lost too. Hence, restarting with a different approach might be the best choice if the prompt extends too much.

We also tried to create prompts using different languages. Although not extensively evaluated, this point did not prove to be very significant for the test case generation in general, but it did impact other relevant aspects. English proved more efficient as it has less room for ambiguity, and prompts tend to be shorter to explain the same scenario than the ones in Portuguese. However, a study on this objective must be conducted to qualitatively assess this issue in future work.

7 Conclusion

The efficiency of ChatGPT in generating test cases for mutation testing stems from its ability to understand complex code structures, generate diverse and meaningful test scenarios, and anticipate possible edge cases. By leveraging its advanced natural language processing capabilities, ChatGPT can analyze program requirements, the codebase, and existing test cases to create comprehensive and targeted test cases that address various aspects of the program's functionality.

However, while ChatGPT excels at automatically generating these test cases, there are still some areas where additional assistance may be necessary to ensure a complete testing process. These areas include:

Seeking to eliminate all mutants: For this study, the live mutants were not made available to the LLM in order to evaluate its efficiency solely by accessing the source code. However, in order to maximize the score and eliminate all mutants, a step could be incorporated into the study where the live mutants become information accessible to the LLM, enabling the generation of highly specific sets and verifying whether the maximum score can be achieved from this.

Analysis of mutation operators: An interesting point that could further this study would be to conduct an in-depth analysis of which mutation operators ChatGPT has the most difficulty generating effective test sets to detect. This would help in understanding and mitigating the challenges it faces.

Understanding specific domain and context: While ChatGPT can generate test cases based on the provided code and requirements, having specific domain knowledge can enhance the relevance and accuracy of the test cases.

Manual validation: Although ChatGPT can automate the generation of test cases, manual validation by a developer or tester is crucial to ensure that the test cases align with the actual behavior and requirements of the program.

Integration with existing testing frameworks: ChatGPT can provide test cases in various formats, but integrating these test cases into existing testing frameworks and ensuring seamless execution may require additional setup and customization.

Information limitations: There are situations where the amount of information needed for a complete understanding of the problem may exceed ChatGPT-3.5's capacity, the subject of this study, to process. To ensure that tests are performed efficiently, human support is needed to break down the code in these cases into smaller, more manageable pieces.

Ongoing updates and maintenance: As the program evolves, test cases need to be updated and maintained to reflect changes. This requires continuous effort and collaboration between development and testing teams.

In summary, ChatGPT is highly efficient in generating test cases for mutation testing, providing a significant boost to the testing process by offering comprehensive and targeted test scenarios. However, to achieve a complete and effective testing process, some assistance is needed in understanding the specific domain, manual validation, integration with existing frameworks, and ongoing updates. By combining the capabilities of LLMs with human expertise, a robust and reliable testing process can be achieved. An interesting direction for future studies would be to increase the number of simultaneous codes analyzed by LLMs, the generation of tests that involve interactions between multiple programs, and even tests where the input of one program depends on the output of another, increasing the complexity of the information that needs to be analyzed.

Bibliography

AMMANN, P.; DELAMARO, M. E.; OFFUTT, J. Establishing Theoretical Minimal Sets of Mutants. In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2014. (ICST '14, v. 1), p. 21–30. ISBN 978-1-4799-2255-0. Citado na página 21.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. Cambridge, Reino Unido: Cambridge University Press, 2008. Citado na página 27.

ANDRADE, S. A. et al. Analyzing the effectiveness of One-Op Mutation against the minimal set of mutants. In: *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing*. New York, NY, USA: Association for Computing Machinery, 2019. (SAST'19), p. 22–31. ISBN 978-1-4503-7648-8. Disponível em: <https://doi.org/10.1145/3356317.3356321>. Citado na página 21.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: *XXVII International Conference on Software Engineering – ICSE'05*. St. Louis, MO, USA: ACM Press, 2005. p. 402–411. ISBN 1-59593-963-2. Citado na página 14.

AWS. *What are Transformers in Artificial Intelligence?* 2024. Last access: 2024/07/18. Disponível em: <https://aws.amazon.com/what-is/transformers-in-artificial-intelligence>. Citado 2 vezes nas páginas 5 and 18.

BARR, E. T. et al. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, v. 41, n. 5, p. 507–525, 2015. Citado 2 vezes nas páginas 15 and 30.

COHEN, M. B. The evolutionary landscape of SBST: a 10 year perspective. In: *Proceedings of the 10th International Workshop on Search-Based Software Testing*. Buenos Aires, Argentina: IEEE Press, 2017. (SBST'17, v. 1), p. 47–48. ISBN 978-1-5386-2789-1. Citado na página 15.

DELAMARO, M. E. et al. Parallel Execution of Programs as a Support for Mutation Testing: A Replication Study. *International Journal of Software Engineering and Knowledge Engineering*, v. 31, n. 03, p. 337–380, 2021. Disponível em: <https://doi.org/10.1142/S0218194021500121>. Citado na página 21.

DELAMARO, M. E. et al. Experimental Evaluation of SDL and One-Op Mutation for C. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. Cleveland, Ohio, USA: [s.n.], 2014. p. 203–212. Citado 6 vezes nas páginas 7, 21, 22, 23, 31, and 32.

DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Proteum/IM 2.0: An Integrated Mutation Testing Environment. In: WONG, W. E. (Ed.). *Mutation Testing for the New Century*. Boston, MA: Springer US, 2001. p. 91–101. ISBN 978-1-4757-5939-6. Disponível em: <https://doi.org/10.1007/978-1-4757-5939-6_17>. Citado 2 vezes nas páginas 21 and 22.

DELAMARO, M. E.; OFFUTT, J.; AMMANN, P. Designing Deletion Mutation Operators. In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2014. (ICST '14), p. 11–20. ISBN 978-1-4799-2255-0. Citado na página 21.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, v. 11, n. 4, p. 34–43, abr. 1978. ISSN 0018-9162. Citado na página 14.

DING, S.; TAN, H. B. K. Detection of Infeasible Paths: Approaches and Challenges. In: MACIASZEK, L. A.; FILIPE, J. (Ed.). *Evaluation of Novel Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 64–78. ISBN 978-3-642-45422-6. Citado na página 15.

DING, S.; TAN, H. B. K.; LIU, K. P. A Survey of Infeasible Path Detection. In: *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. Wroclaw, Poland: SciTePress, 2012. p. 43–52. ISBN 978-989-8565-13-6. Citado na página 15.

ELOUNDOU, T. et al. *GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models*. 2023. Disponível em: <https://openai.com/research/gpts-are-gpts>. Citado na página 16.

FRASER, G.; ARCURI, A. 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: ACM, 2011. (ESEC/FSE '11, v. 1), p. 416–419. ISBN 978-1-4503-0443-6. Citado na página 15.

FRASER, G.; ARCURI, A. EvoSuite at the SBST 2016 Tool Competition. In: *Proceedings of the 9th International Workshop on Search-Based Software Testing*. Austin, Texas: ACM, 2016. p. 33–36. ISBN 978-1-4503-4166-0. Citado na página 15.

GeeksforGeeks. *Large Language Model (LLM)*. 2024. Acesso em: 04 jul. 2024. Disponível em: <https://www.geeksforgeeks.org/large-language-model-llm/>. Citado na página 16.

GUILHERME, V. H.; VINCENZI, A. M. R. An initial investigation of ChatGPT unit test generation capability. In: *8th Brazilian Symposium on Systematic and Automated Software Testing – SAST'2023*. Campo Grande, MS: ACM Press, 2023. p. 15–24. Citado 3 vezes nas páginas 19, 20, and 34.

HURLEY, E.; OKYERE-BADOO, J. A Comparative Study of Few-Shot vs. Zero-Shot Prompting to Generate Quick and Useful Responses to Students' Periodic Reflections. In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*. New York, NY, USA: Association for Computing Machinery, 2024. (SIGCSE 2024), p. 1881. ISBN 9798400704246. Disponível em: <https://doi.org/10.1145/3626253.3635400>. Citado na página 20.

JATANA, N.; SURI, B. An Improved Crow Search Algorithm for Test Data Generation Using Search-Based Mutation Testing. *Neural Processing Letters*, v. 52, n. 1, p. 767–784, ago. 2020. ISSN 1573-773X. Citado na página 15.

JURAFSKY, D.; MARTIN, J. H. *Speech and Language Processing (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009. 83-122 p. ISBN 0131873210. Citado na página 16.

KUSHARKI, M. B. et al. Automatic Classification of Equivalent Mutants in Mutation Testing of Android Applications. *Symmetry*, v. 14, n. 4, p. 14, 2022. ISSN 2073-8994. Citado na página 15.

LI, T.-O. et al. *Finding Failure-Inducing Test Cases with ChatGPT*. 2023. Citado na página 19.

LIU, Z. et al. *Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing*. 2023. Disponível em: <https://doi.org/10.48550/arXiv.2305.09434>. Citado na página 20.

LUKASCZYK, S.; FRASER, G. Pynguin: automated unit test generation for Python. In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2022. (ICSE'22), p. 168–172. ISBN 978-1-4503-9223-5. Disponível em: <https://doi.org/10.1145/3510454.3516829>. Citado na página 15.

MA, W. et al. *The Scope of ChatGPT in Software Engineering: A Thorough Investigation*. 2023. Disponível em: <https://doi.org/10.48550/arXiv.2305.12138>. Citado na página 19.

NETO, J. C. et al. A Strategy to Support the Infeasible Test Requirements Identification. In: *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing*. New York, NY, USA: Association for Computing Machinery, 2022. (SAST '22, v. 1), p. 29–38. ISBN 978-1-4503-9753-7. Citado na página 15.

NGUYEN, C. et al. An Approach to Generating API Test Scripts Using GPT. In: *Proceedings of the 12th International Symposium on Information and Communication Technology*. New York, NY, USA: Association for Computing Machinery, 2023. (SOICT '23, v. 1), p. 501–509. ISBN 9798400708916. Citado na página 16.

OZKAYA, I. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software*, v. 40, n. 3, p. 4–8, 2023. Citado na página 16.

PACHECO, C.; ERNST, M. D. Randoop: Feedback-directed Random Testing for Java. In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. Montreal, Quebec, Canada: ACM, 2007. (OOPSLA '07), p. 815–816. ISBN 978-1-59593-865-7. Citado na página 15.

PAPADAKIS, M. et al. Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship between Mutants and Real Faults. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. (ICSE'18), p. 537–548. ISBN 978-1-4503-5638-1. Event-place: Gothenburg, Sweden. Disponível em: <https://doi.org/10.1145/3180155.3180183>. Citado na página 14.

PIYA, S.; SULLIVAN, A. {LLM4TDD}: Best Practices for Test Driven Development Using Large Language Models. In: *1st International Workshop on Large Language Models for Code – LLM4Code'2014*. Lisboa, Portugal: [s.n.], 2024. p. 8. Disponível em: <https://arxiv.org/abs/2312.04687>. Citado na página 16.

RASNAYAKA, S. et al. An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project. In: *1st International Workshop on Large Language Models for Code – LLM4Code'2014*. Lisboa, Portugal]: [s.n.], 2024. p. 8. Disponível em: <https://arxiv.org/abs/2401.16186>. Citado na página 16.

RIBEIRO, M. T. Página Web, *Testing Language Models (and Prompts) Like We Test Software*. 2023. Disponível em: <https://towardsdatascience.com/testing-large-language-models-like-we-test-software-92745d28a359>. Citado na página 20.

ROJAS, J. M. et al. A Detailed Investigation of the Effectiveness of Whole Test Suite Generation. *Empirical Softw. Engg.*, v. 22, n. 2, p. 852–893, abr. 2017. ISSN 1382-3256. Place: USA Publisher: Kluwer Academic Publishers. Citado na página 15.

ROPER, M. *Software Testing*. New York, NY: McGrall Hill, 1994. Citado 2 vezes nas páginas 13 and 14.

SIDDIQ, M. L. et al. *Exploring the Effectiveness of Large Language Models in Generating Unit Tests*. 2023. Disponível em: <https://doi.org/10.48550/arXiv.2305.00418>. Citado na página 19.

SILVA, R. A.; SOUZA, S. d. R. S. d.; SOUZA, P. S. L. d. A systematic review on search based mutation testing. *Information and Software Technology*, v. 81, p. 19–35, 2017. ISSN 0950-5849. Citado na página 15.

SINGH, A. et al. A Systematic Literature Review on Test Case Prioritization Techniques. In: *Agile Software Development*. New Jersey, NY: John Wiley & Sons, Ltd, 2023. p. 101–159. ISBN 978-1-119-89683-8. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119896838.ch7>. Citado na página 15.

TAHBILDAR, H.; BORBORA, P.; G.P., K. Teaching automated test data generation tools for c, c++ , and java programs. *International Journal of Computer Science and Information Technology*, v. 5, p. 181–195, 02 2013. Citado na página 15.

TANG, Y. et al. *ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation*. arXiv, 2023. Disponível em: <http://arxiv.org/abs/2307.00588>. Citado na página 19.

VASWANI, A. et al. Attention is All You Need. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2017. (NIPS'17, v. 1), p. 6000–6010. ISBN 978-1-5108-6096-4. Event-place: Long Beach, California, USA. Citado na página 19.

WEIZENBAUM, J. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 9, n. 1, p. 36–45, jan 1966. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/365153.365168>. Citado na página 16.

WHITE, J. et al. *ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design*. 2023. Disponível em: <https://doi.org/10.48550/arXiv.2303.07839>. Citado na página 19.

WINOGRAD, T. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. 1971. Citado na página 16.

XIE, Z. et al. *ChatUniTest: a ChatGPT-based automated unit test generation tool*. 2023. Disponível em: <https://doi.org/10.48550/arXiv.2305.04764>. Citado na página 19.

YONG, G. et al. Prompt engineering for zero-shot and few-shot defect detection and classification using a visual-language pretrained model. *Computer-Aided Civil and Infrastructure Engineering*, v. 38, n. 11, p. 1536–1554, 2023. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1111/mice.12954>. Citado na página 20.

YUAN, Z. et al. *No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation*. 2023. Disponível em: <https://doi.org/10.48550/arXiv.2305.04207>. Citado na página 19.