Improving the Readability of Automatically Generated Tests using Large Language Models

Matteo Biagiola, Gianluca Ghislotti, Paolo Tonella
Software Institute - Università della Svizzera italiana, Lugano, Switzerland
{matteo.biagiola, gianluca.ghislotti, paolo.tonella}@usi.ch

***Abstract*—Search-based test generators are effective at producing unit tests with high coverage. However, such automatically generated tests have no meaningful test and variable names, making them hard to understand and interpret by developers. On the other hand, large language models (LLMs) can generate highly readable test cases, but they are not able to match the effectiveness of search-based generators, in terms of achieved code coverage.**

In this paper, we propose to combine the effectiveness of search-based generators with the readability of LLM generated tests. Our approach focuses on improving test and variable names produced by search-based tools, while keeping their semantics (i.e., their coverage) unchanged.

Our evaluation on nine industrial and open source LLMs show that our readability improvement transformations are overall semantically-preserving and stable across multiple repetitions. Moreover, a human study with ten professional developers, show that our LLM-improved tests are as readable as developer-written tests, regardless of the LLM employed.

***Index Terms*—Large Language Models, Software Testing, Readability.**

I. Introduction

Automated test generation techniques have been studied in depth in the last decades. Such techniques are very appealing as they promise to automate the test input creation process, easing the burden on software developers.

One of the prominent ways the test input creation process can be automated is by modelling the problem via the search-based framework [1]. This methodology employs search and optimization algorithms to automatically generate test cases that maximize a certain objective, e.g., code coverage and/or bug detection, providing an efficient and scalable solution [2]. Unit test generation for Java programs is one of the most active research areas in this field, with state-of-the-art tools such as `Evosuite` [3], whose core generation technique has been improved over the years [4], [5], [6]. Moreover, search-based techniques have been applied to other programming languages such as Python [7], and Javascript [8], as well as for system level testing of RESTful APIs [9], Web and Android applications [10], [11]. Despite their effectiveness, search-based generators have been criticized as they generate tests with low readability, making them hard to interpret (e.g., when checking the oracle, diagnosing the failures or understanding/-documenting their behaviour), and maintain [12], [13].

Recently, large language models (LLMs) have been proposed as a way to address the software test creation problem. The work of Tufano et al. [14] pioneered this research subfield, by formalizing the test generation problem as a sequence-to-sequence translation problem. In a nutshell, `AthenaTest` exploits a Transformer architecture [15] to train a language model that would translate a method under test (i.e., a focal method) into its corresponding test case. More recent approaches use pre-trained LLMs. They supply LLMs a carefully crafted prompt, which may include input/output examples (using LLMs in *few-shot* mode), to generate test cases given the unit under test as context. Examples of test generators belonging to this category include `TestPilot` [16], a few-shot LLM-based test generator for Javascript, `ChatUnitTest` [17] and `ChatTester` [18], proposing an LLM-based generation-validation-repair framework for generating Java unit tests, and `TestSpark` [19], a plugin for the `IntelliJ` IDE that supports LLM-based test generation. Multiple empirical studies in the literature show that tests generated by LLMs are more readable than tests generated by `Evosuite` [14], [20], as developers tend to prefer them to tests generated via coverage-guided methods. On the other hand, `Evosuite`'s tests are more effective than LLM generated tests, in terms of coverage and bug detection [21], [22], [20].

In summary, the literature suggests that coverage-guided test generators such as `Evosuite` generate effective test cases, while LLM-based generators generate more readable but less effective tests. The objective of our work is to combine the best of both worlds, i.e., the readability of LLM-based test suites while maintaining the performance of test suites generated through search-based methods. In particular, we propose to use LLMs to improve the readability of `Evosuite` tests. While existing LLM-based approaches aim to refactor the entire structure of test cases to improve their readability [23], [24], we focus instead on identifiers and test names, as previous studies in the literature suggest that these have a large influence on the test's readability [12], [13]. Moreover, existing approaches do not explicitly optimize the input prompt to the LLM. Indeed, LLMs have a fixed context window, that long classes/tests may quickly saturate. Large classes/test suites, would also result in long prompts that have been shown to be detrimental for LLMs as they give rise to *lost-in-the-middle* effects [25], where LLMs lose their capability to meaningfully attend relevant information in the middle of the prompt. We,

XXX-X-XXXX-XXXX-X/25/$31.00 ©2025 IEEE
DOI XX.XXXX/ICSTXXXXX.2025.XXXXX
2025 IEEE Conference on Software Testing, Verification and Validation (ICST)

instead, design a multi-step prompt, where we first feed the LLM the focal information of the class under test (i.e., class name, constructors, attributes and method signatures), which is kept in memory by the LLM and prefixed to each subsequent request. We then provide the LLM with each individual test to be improved, together with the method bodies of the class under test it exercises. Such individual prompts are submitted independently from each other, to keep the overall prompt short and the context window limited.

We evaluated our approach using nine industrial and open-source LLMs. Our results show that most LLMs are able to preserve the semantics of the tests (i.e., their coverage) while improving their readability. Moreover, the readability improvements of the considered LLMs are quite stable across repetitions, despite the potential non-determinism of their output. We also conducted a human study with ten professional developers who evaluated the readability of LLM-improved tests w.r.t. developer-written tests. Results show that LLM-improved tests are equally readable as developer-written tests, regardless of the LLM used. The reliability of our stable and semantically-preserving readability transformations, as well as the comparable readability w.r.t. developer-written tests, make our approach a viable tool to be used in practice to improve the readability of coverage-guided automatically generated tests.

II. Related Work

A. Automated Test Generation

Search-based Techniques. The topic of Search-based software testing has a rich literature, where numerous surveys and empirical studies have been conducted [26], [27], [28]. The state-of-the-art tool for the generation of Java unit test cases is the search-based test generator Evosuite [3], although Randoop [29], a *feedback-directed* random test generator, is often used in the literature especially as a strong baseline in Java tool competitions [30]. Beyond Java unit test generation, search-based techniques are also used by researchers to generate test cases for RESTful APIs [31] (with the tool Evomaster [9]), Android GUI test cases [11] (with the tool Sapienz) and unit tests for Python [32] (with the tool Pynguin [7]). Search-based techniques are very effective at generating high-coverage tests, also given their ability to focus the search on multiple areas of the search space. Indeed, one of the most recent innovation is to target all the coverage goals at once, either by aggregating the fitness of each coverage goal in a single fitness function [4] or by formulating the search problem as a many-objective optimization problem [6], [5], which seems to be more effective (indeed, DynaMOSA [6] is the default search algorithm in Evosuite).

LLMs-based Test Generation. The use of LLMs in software testing is relatively new. However, from January 2019 to October 2023, there have been more than 100 publications related to the use of LLM in software testing (more than 80% in 2023) in both software engineering and AI venues [33]. The survey by Wang et al. [33] summarizes the state of the art on this topic, discussing the use of LLMs for software testing activities, such as unit and system test generation, oracle generation, debugging, and program repair.

Tufano et al. [14] describe an approach implemented in the tool AthenaTest, which generates test cases by solving a *sequence-to-sequence translation* problem (the tool A3Test by Alagarsamy et al. [34] is also an instance of this test generation class). The authors use an encoder-decoder transformer model (i.e., BART), pre-train it on a large corpus of English and Java source code, and finetune it using a translation task, where the source language is a *focal* method (i.e., the method under test), and the target is the unit test written by a human developer for that method.

More recent approaches simplify the test generation problem, by employing LLMs with *zero-shots* or *few-shots* prompts. Schafer et al. [16] propose TestPilot, a few-shot LLM-based test generator for the API of a given Javascript project. Plein et al. [35] use bug reports as prompt to a language model (in particular ChatGPT and codeGPT) to generate executable test cases. Siddiqa et al. [21] conduct an empirical study on the effectiveness of three language models, namely gpt3.5-turbo, Codex, and StarCoder for generating unit tests for Java. Chen at al. [17], [22] introduce a framework named ChatUnitTest for generating Java unit tests. The framework consists of a generation-validation-repair mechanism to fix errors in the generated unit test; the validation check runs a Java parser, compiles the code and runs it. Tang et al. [20] statistically compare ChatGPT with Evosuite w.r.t. statement coverage on the SF110 dataset, and w.r.t. bug detection on Defects4J projects. Results show that Evosuite achieves significantly higher statement coverage than ChatGPT (i.e., 77% vs 55% on average), and exposes more bugs (i.e., 55 bugs vs 44 on average). In terms of readability, which the authors measure quantitatively using code style standards as well as *cyclomatic complexity* [36] and *cognitive complexity* [37], ChatGPT test cases do not seem to adhere to a specific code style, while they feature a low complexity that make them easy to follow. Similarly, Yuan et al. [18] propose ChatTester, a ChatGPT-based generator that features an intention prompt, to understand the focal method (i.e., the method under test), and a generation prompt, to generate a test for such method.

Hybrid Techniques. Researchers have also tried to combine LLMs with traditional software testing techniques to generate more effective test cases. For instance, Lemieux et al. [38] use LLMs within the evolutionary loop of DynaMOSA to generate the right data to get the search unstuck. Similarly, Arghavan et al. [39] use mutation testing to improve the effectiveness of test cases generated by LLMs. Such approaches do not specifically focus on readability but rather on either supporting the search-based generators in specific situations [38], or guiding the generation process of LLMs [39].

B. Readability

Code Readability. Buse et al. [40] construct an automated readability metric by building a classifier that predicts human readability by using a simple set of local code features.

Similarly, Campbell et al. [37] propose a cognitive complexity metric, designed to address the shortcomings of cyclomatic complexity, such as the nesting problem. Munoz et al. [41] systematically evaluated whether this metric actually captures source code understandability. Results show that cognitive complexity is a promising metric to automatically assess different aspects of code understandability, although code understandability can be measured in different ways (e.g., whether the comprehension task is completed successfully [42], time to locate and fix a bug [43], or perceived understandability [44]), and it is not yet clear how those ways are related.

Test Readability Improvement. Regarding software tests, researchers have focused specifically on improving the readability of automatically generated tests. Daka et al. [12] propose a readability metric for Java unit tests. The authors train a linear regressor on human annotated data to predict readability scores from a set of features. Panichella et al. [45] propose `TestDescriber`, an approach that automatically generates a summary of the source code exercised by a certain test case. Daka et al. [13] focus on an automated approach to generate descriptive test names, based on the functionalities they cover. Roy et al. [46] propose `DeepTC-Enhancer` to improve the readability of automatically generated tests. The approach works in two steps: first it generates method-level summaries for given test cases; then, it resorts to code summarization by training a deep learning model to carry out the tasks of test name and variable name predictions. The objective of the second step is to rename all the identifiers and the test name. Delgado et al. [47] adopt a different approach to test readability. In particular, they integrate readability assessments within the evolutionary loop of `Evosuite`, with the purpose of generating test cases that are more readable according to the tester's preferences. The tester gives a readability score to each test case, which the evolutionary loop takes into account to decide which tests to keep for the next generations (together with the coverage goals).

The works described above mostly adopt machine/deep learning techniques or custom heuristics to address the readability problem in automatically generated tests. Such models need to be trained on datasets of human preferences. Given the cost of labeling, the size of such datasets is limited, potentially failing to capture all the factors that might affect the subjective readability evaluations. We instead resort to LLMs that are trained on massive corpora of source and test code, to tackle the readability task with the knowledge of many more developers. Moreover, while custom heuristics focus only on specific aspects of readability (e.g., the test names), LLMs can target the readability problem more holistically. The works of Gay et al. [48] and Alshahwan et al. [24] are more related to our approach, as they both use LLMs to improve test cases. The focus of Gay et al. [48] is to improve Python unit tests automatically generated by `Pynguin`, while Alshahwan et al. [24] propose and evaluate a tool named `TestGen-LLM` to improve Kotlin test cases manually written for Meta's products. Despite the common objective, our work differs in several fundamental ways: (1) our prompt is designed not to change the semantics of the test, contrary to `TestGen-LLM` [24] that specifically aims to increase the coverage of the existing test cases, and Gay et al. [48], whose approach potentially introduces semantic changes; (2) we deal with the limited context window size of an LLM by designing a multi-step prompt; (3) we take into account the randomness of the LLM (even at temperature zero [49]) on the readability transformations, by analyzing how much the transformations vary across runs; (4) we evaluate the performance of nine industrial and open-source LLMs, while Gay et al. [48] only consider `gpt-4` and Alshahwan et al. [24] evaluate two LLMs internally-developed at Meta.

III. Motivating Example

Listing 1 shows the `Stack` class, that we use as a motivating example. A stack has two basic functionalities, i.e., `push` and `pop`. In this example, a stack has a capacity of 3, and has no resize functionality, i.e., once the stack is full it is not possible to add new elements.

```
public class Stack<T> {
    private int capacity = 3;
    private int pointer  = 0;
    private T[] objects = (T[]) new Object[capacity];
    public void push(T o) {
        if(pointer >= capacity)
            throw new RuntimeException("Stack exceeded
                 capacity!");
        objects[pointer++] = o;
    }
    public T pop() {
        if(pointer <= 0)
            throw new EmptyStackException();
        return objects[--pointer];
    }
}
```

Listing 1: Class implementing the stack data structure.

The `Stack` class has five branches: a branch in class constructor (not shown), two branches in the `push` method, corresponding to the condition `pointer >= capacity` being `true` or `false`, and two branches in the `pop` method for the condition `pointer <= 0`. A unit test generator for this class might generate a test suite similar to the one below:

```
public class Stack_ESTest {
  @Test
  public void test0()  throws Throwable  {
      Stack<Integer> stack0 = new Stack<Integer>();
      try {
        stack0.pop();
        fail("Expecting exception: EmptyStackException");
      } catch(EmptyStackException e) {
         verifyException("tutorial.Stack", e);
      }
  }
  @Test
  public void test1()  throws Throwable  {
      Stack<Integer> stack0 = new Stack<Integer>();
      Integer integer0 = new Integer(0);
      stack0.push(integer0);
      stack0.push(integer0);
      stack0.push(integer0);
      try {
        stack0.push(integer0);
        fail("Expecting exception: RuntimeException");
      } catch(RuntimeException e) {
         verifyException("tutorial.Stack", e);
      }
  }
  @Test
  public void test2()  throws Throwable  {
      Stack<Object> stack0 = new Stack<Object>();
      Object object0 = new Object();
      stack0.push(object0);
      Object object1 = stack0.pop();
      assertSame(object1, object0);
  }
}
```

Listing 2: Test suite for the `Stack` class generated by `Evosuite` with the branch coverage criterion. The test suite covers all five branches of the class under test.

In this example, we used `Evosuite` to generate the test suite in Listing 2, a state-of-the-art search-based unit test generator for Java. In particular, we used the branch coverage criterion, such that the test generator has the objective to cover all the branches of the `Stack` class. While all branches are covered, tests are hard to read, especially because test names do not convey any information on what the underlying test does. Our approach aims to improve the readability of such automatically generated tests, while keeping their coverage effectiveness intact.

IV. Approach

Figure 1 shows an overview of our approach for test case readability improvement. Our approach takes as input the class under test; then employs a test generator to automatically generate a test suite. From the test suite it extracts the individual test cases, together with the corresponding *focal context* of the class under test, i.e., the functionalities of the class exercised by each test case. These two types of information make up the prompt of the LLM, which modifies each single test case and improves the readability of the generated test suite.

A. Prompt Design

In designing our prompts, we follow the `OpenAI` prompt engineering guidelines [1], as well as established results coming from the literature [50]. Our *prompt engineering* strategy consists of splitting the readability task into multiple subtasks[2], where we first provide the context the LLM needs to improve readability, i.e., the class under test and the goal we want to achieve, and then we feed each single test case of the test suite that needs to be transformed. The reason is twofold: (1) providing all available information in a single prompt, i.e., class under test and the whole test suite, and asking the LLM to improve the readability of the test suite, would be an overly complex task that could hinder the quality of the output. Moreover, long prompts can be affected by *lost-in-the-middle* effects [25], where the model's performance degrades when it needs to access information in the middle of long prompts; (2) LLMs have a limited context window size, measured in the number of tokens making up the input prompt. If the class under test is long, and/or the test suite contains several test cases (or few long test cases), then the input would not fit the context window size of the model, and some relevant information would be discarded. In summary, by designing a multi-step prompt, we both reduce the size of each prompt, in terms of number of tokens, and we focus the attention of the LLM on smaller tasks, avoiding long prompts.

The first prompt that we send to the LLM is the *Informational prompt* (shown below). The goal of this initial prompt is to clearly state the objective of the readability task, i.e., improving the readability of test cases by only modifying the identifiers and the test names. We also specify that the readability task will be broken down into multiple steps, one per test case. To keep the size of the prompt manageable, we only include the essential information of the class under test in the initial prompt, i.e., the class name, the constructors (including their bodies), the attributes (both private and public), and the signatures of the methods (both private and public). We call such essential information *focal context*, inserted into the prompt by replacing the content inside braces "$\{sc\}$". We configured the LLM to *memorize* this first prompt, such that it is prefixed to the prompts of each subsequent request.

Informational prompt

You are a professional Java programmer.

Your ultimate goal is to improve the readability of the test cases I will send you, particularly by modifying ONLY the identifiers, and the test name, NOT THE METHODS CALLED INSIDE THE TESTS, STATIC METHODS, OR STATIC CLASSES.

Thinking in steps:

1. Initially (this prompt), I will send you the original source code of the class under test, to give you the context and the aim of the class.

2. In the next prompts, I will send you each single test of the test suite you need to improve the readability of, as well as the source code of the original class methods that are called in the test.

Focal information of the class under test:

$\{sc\}$

The prompt starts with asking the model to adopt a persona[3], in this case a *professional Java programmer*. The second strategy is the use of words in capital letters to emphasize that only the identifiers and test names should be modified; indeed, we noticed from preliminary experiments that without such emphasis the LLM tends to modify method names as well. Moreover, we use the natural language statement "Thinking in steps" to explicitly encourage the model to follow our multi-step task. This prompt strategy is known as *Chain-of-Thought* (i.e., CoT for short), and it is the foundation of all strategies related to reasoning and logic tasks. Indeed, Wei at al. [51] propose such technique to induce a step-by-step reasoning

[1] Prompt engineering guidelines (Accessed August 2024)

[2] Split tasks into subtasks (Accessed August 2024)

[3] Ask model to adopt persona (Accessed August 2024)

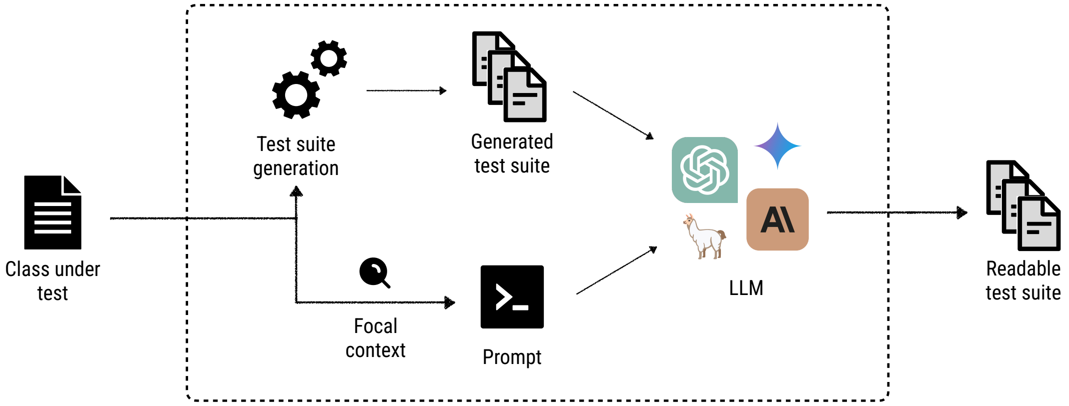

Fig. 1: Overview of our approach to readability improvement. A test generator produces the test suite given the class under test taken as input. The *focal context* of the class under test, as well as the generated test cases, make up the prompt of the LLM, improving the readability of the generated test suite.

process in the LLM, by manually detailing the logical steps such that the LLM can replicate them while solving the task. Kojima et al. [52] found that using the phrasing "Let's think step by step" in the prompt would induce a similar behavior, without the cost of manually detailing the reasoning process. Finally, we use delimiters (colon and following new line) to separate the source code from the rest of the prompt[4].

Improvement prompt

Improve the readability of the test below by modifying ONLY the identifiers, test name and variable names, NOT THE METHODS CALLED INSIDE THE TESTS, STATIC METHOD OR CALLED STATIC CLASS. The changes must not affect the functioning of the test in any way.

Test to modify:

{*single_test*}

Knowing the source code of all the methods used in the test:

{*sc_test_calls*}

Answer with code only.

The box above shows the second prompt, which we call *Improvement prompt*. We send such prompt to the LLM for each test case in the generated test suite. By design, improvement prompts are *disjoint*, i.e., each improvement prompt starts a new session, discarding all the previous improvement interactions. In this way, we control the size of the prompt, which includes in each session the informational prompt and the improvement prompt. Hence, the LLM focuses on improving the readability of one test at a time. The improvement prompt has three main sections: the instruction section, which instantiates the instructions in the informational prompt in the context of the current test, the source code of the current test, and the source code of the class under test methods exercised by the test. Although our prompt design makes it difficult to reach the limit of the context window for the current LLMs, long classes and/or long test methods might still trigger a context overflow. In such situations, we proceed by first, discarding the source code of the methods exercised by the test in the improvement prompt; then, we remove the informational prompt form the memory, focusing only on improving the current test without context. In the rare cases where the improvement prompt with just the inclusion of the test to modify does not fit the prompt size, we give up improving the test and report it as-is in the final test suite.

Remove duplicates prompt

These tests have the same names:

{*tests*}

Change them such that they differ, while their objectives remain clear. The content of the tests must remain exactly identical.

Answer with code only.

Finally, we check that all the tests in the final test suite have unique names. Indeed, since each improvement prompt is separated from the others, it may happen that the LLM assigns the same name to tests that are structurally or semantically similar. The *Remove Duplicates Prompt* (shown above), is designed to disambiguate test names in case of duplicates. We simply provide the LLM the source code of the test cases that have the same names, and ask it to change them while keeping their content unchanged. We submit the remove duplicates prompt for each set of tests that have the same name, and

[4] Use delimiters (Accessed August 2024)

proceed to query the LLM until all ambiguities are fixed (we keep track of the number of times we query the LLM, and fall back to the original test names after 3 queries). Finally, we output the improved test suite at the end of the process.

B. *Motivating Example*

Let us apply the proposed approach to improve the readability of the `Stack` test suite generated by `Evosuite` (Listing 2). The first step, is to extract the focal context from the `Stack` class to build the informational prompt. In this case, we just copy the class in the prompt, without the bodies of the methods `push` and `pop`, replacing the "{*sc*}" tag in the informational prompt. Next, for each test in the test suite, we build an improvement prompt. For instance, to improve `test1`, we replace the "{*single_test*}" tag in the improvement prompt with the source code of `test1`, and the "{*sc_test_calls*}" tag with the source code of the `push` method of the `Stack` class.

The resulting test suite after running our approach is shown in Listing 3 (we used one of the LLMs considered in the evaluation). We observe that test names are now meaningful and reflect what the test actually does, making the test suite more understandable and maintainable. For instance, `test1` is called `testPushCapacityExceeded` in the improved test suite, as the test indeed exercises a stack overflow. We also notice that the LLM did not change the semantics of the test cases, as it safely modified the test names and the identifiers within each test, without altering any execution flow or introducing errors.

```java
@Test
public void testPopOnEmptyStack() throws Throwable {
    Stack<Integer> integerStack = new Stack<Integer>();
    try {
        integerStack.pop();
        fail("Expecting exception: EmptyStackException");
      } catch(EmptyStackException e) {
         // no message in exception (getMessage() returned
              null)
         verifyException("tutorial.Stack", e);
      }
}
@Test
public void testPushCapacityExceeded() throws Throwable {
    Stack<Integer> integerStack = new Stack<Integer>();
    Integer inputInteger = new Integer(0);
    integerStack.push(inputInteger);
    integerStack.push(inputInteger);
    integerStack.push(inputInteger);
    try {
        integerStack.push(inputInteger);
        fail("Expecting exception: RuntimeException");
      } catch(RuntimeException e) {
         // Stack exceeded capacity!
         verifyException("tutorial.Stack", e);
      }
}
@Test
public void testPopReturnsPushedObject() throws Throwable {
    Stack<Object> objectStack = new Stack<Object>();
    Object pushedObject = new Object();
    objectStack.push(pushedObject);
    Object poppedObject = objectStack.pop();
    assertSame(poppedObject, pushedObject);
}
```

Listing 3: Test suite for the `Stack` class generated by `Evosuite` and improved by our approach.

V. Empirical Evaluation

To assess the practical benefits of our readability improvement approach, we formulate the following research questions:

RQ$_1$ (semantic preservation). *To what extent is the semantics of a test case preserved after readability improvements?*
RQ$_2$ (stability). *To what extent are readability improvements stable across multiple repetitions?*
RQ$_3$ (human study). *How do developers judge the readability of LLM-improved test cases w.r.t. developer-written test cases?*

RQ$_1$ and RQ$_2$ aim to assess whether LLMs are *reliable* in the readability improvement task, making them a useful tool that can be used in practice. In particular, RQ$_1$ analyzes the extent to which the LLM changes the semantics of a test case across multiple repetitions. Although our prompt is designed not to change the semantics of a test case, the LLM might ignore the instructions and change, for instance, method calls within the test, leading to compilation errors or to exercise different execution paths of the class under test w.r.t. the original test. We promoted to the next research questions only the LLMs that never change the semantics of a test case across multiple repetitions. In RQ$_2$ we study the *stability* of the readability improvements across multiple repetitions. In other words, we analyze how much the readability improvements vary when the LLM is prompted to change the same test case multiple times. In RQ$_3$, we ask developers to compare the readability of automatically generated test cases improved by LLMs, with that of developer-written test cases. Our objective with this human study is to show that LLM-improved test cases get close to the readability of developer-written test cases. Since readability is hard to measure automatically, and eventually test cases are used and interpreted by developers, we designed a human study to capture the perceived readability.

A. *Classes Selection*

The first step of our empirical evaluation is the selection of the classes under test. We selected classes from five well known Java projects, i.e., *Apache Commons Lang* (Lang henceforth)[5], *JFreeChart* (Chart henceforth)[6], *Apache Commons Cli* (Cli henceforth)[7], *Apache Commons Csv* (Csv henceforth)[8], and Google *Gson* (Gson henceforth)[9]. We rely on these projects because they are equipped with high quality manual test suites, which we use in the human study.

Since our ultimate goal is to evaluate the readability of test cases through a human study, we selected two classes per project, for a total of ten classes, in order to keep the study's size manageable. To select the classes, we first extracted from each project, and for each class within each project, the lines of code (*LOC*), the average method length (*AML*), in terms of number of statements, and the number of internal

[5] https://commons.apache.org/proper/commons-lang (August 2024)
[6] https://jfree.org/jfreechart (August 2024)
[7] https://commons.apache.org/proper/commons-cli (August 2024)
[8] https://commons.apache.org/proper/commons-csv (August 2024)
[9] https://github.com/google/gson (August 2024)

	# Classes			
	Original	Filtering	Core Selection	Readability Study
Cli	25	12	5	2
Csv	11	4	4	2
Lang	246	67	6	2
Gson	73	23	6	2
Chart	657	301	6	2

TABLE I: Classes selection. The table shows the number of classes remaining after each step of the selection process, except the first column ("Original") that shows the number of classes in each project.

imports (*II*), i.e., the number of classes within the respective project that are used by the specific class as dependencies. We then kept classes satisfying the following criteria: (1) not a class with a private/protected constructor and not an abstract class, (2) $LOC \in [50, \inf]$, (3) $AML \in [3, 20]$, (4) $II \in [0, 10]$. The first criterion ensures that a test generator (e.g., Evosuite) can generate tests for those classes. The second filters out trivial classes, while the third takes care of discarding classes with no relevant functionalities (e.g., a data class with only getters and setters would have $AML = 1$). We set an upper bound of 20 statements for the *AML* to exclude classes with long methods, which are likely more difficult to test and inspect than classes with shorter methods. The fourth filter ensures that the resulting classes are relatively self-contained, and do not heavily rely on other internal classes, making them easier to test and inspect. Since the number of classes was still high for some projects after applying the filters (see Column 2 of Table I), we ranked the remaining classes in ascending order based on the number of internal imports (we used *LOC* as a secondary sorting criterion). The first two authors then independently labeled each class as "core"/"non-core", to make sure that the selected classes are classes implementing core functionalities related to the project. We then selected the first six core classes (or less if after filtering the project had less classes) according to the ranking, where there was an agreement between the authors, and we made sure that Evosuite, the test generator we used to automatically generate tests for the selected classes, was able to generate a test suite for such classes. The number of classes after core selection is shown in Column 3 of Table I.

At this stage, we conducted a small pilot study with three PhD students from our lab, to select the two classes out of those resulting from the core selection. The objective of this pilot study was to select the two most *readable* classes from a testing perspective. In the survey used for the pilot study we defined readability as the ability of a developer, different from the one who wrote the class, to design a set of test cases that adequately cover the functionalities of the class. We recommended the three PhD students to spend a maximum of five minutes to evaluate each class, and we asked them to assign a readability score, from 1 (not at all readable) to 5 (very readable), to each class within the same project. For each project, we then summed the scores for each class, and selected the two classes with the highest scores. The selected classes are available in our replication package [53], as well as the printout of the pilot study.

B. Models Selection

We carried out the selection of the large language models to be used for test case readability improvement in April 2024, considering two models from the following providers: OpenAI, Anthropic, Mistral, and Meta, and one model from Google, since only one model was available at the time of the selection. Regarding Anthropic, Mistral, and Meta, we selected the models available on Amazon Bedrock[10], which offers API access to Claude, Llama3 and Mixtral, respectively. Regarding OpenAI and Google, we used the APIs provided by the respective providers, to access GPT and Gemini respectively. In total we selected nine models, i.e., gpt3.5-turbo, and gpt-4 from OpenAI, gemini-1.5-pro from Google, llama-3-8b and llama-3-70b from Meta, claude-3-haiku and claude-3-sonnet from Anthropic, and mistral-7b and mistral-8x7b from Mistral.

C. RQ_1 (semantic preservation)

1) Metrics: Our study on the preservation of the test semantics after improvement is based on the reports generated by Jacoco[11], an open-source toolkit for measuring and reporting Java code coverage. These reports contain indicate how many statements, branches, methods, and lines, were covered when running the tests. The reports provide a comprehensive and detailed overview of code coverage, allowing for an in-depth analysis of which parts of the project have been effectively tested. We define as *success rate* the number of times the readability transformations performed by a given LLM are coverage-preserving, divided by the total number of repetitions. In other words, we approximate semantic-preservation with coverage-preservation.

2) Procedure: For each project, and for each class, we ran Evosuite with the default parameters to generate a test suite for the related classes. Then, for each LLM, and for each generated test suite, we executed our readability improvement approach ten times, to cope with the non-determinism of LLMs [49]. For the LLMs requiring a temperature parameter, we set such parameter to 0, to get more deterministic results. Then, we compared the Jacoco reports when running the improved tests and those produced when running the original Evosuite tests. If the reports are identical, then the readability transformation is deemed successful.

D. RQ_2 (stability)

1) Metrics: We measured stability by computing the *code embeddings* of the tests after readability improvement. Code embeddings are numerical representations of programs, designed to capture both the formal semantics (e.g., syntactical

[10] https://aws.amazon.com/bedrock/ (September 2024)
[11] https://www.eclemma.org/jacoco/ (September 2024)

structure) and the informal semantics (e.g., identifier naming) of the input code, used in tasks like code search and code completion. We embedded a pair of improved tests, i.e., t_i and t_j, into a vector space, and we computed the *cosine similarity* between such vectors. The cosine similarity varies between -1 and 1, and the closer it is to 1, the more similar the two test cases are. In particular, in this context t_i and t_j result from two different repetitions of our readability improvement approach on the same test, as we want to measure how stable the readability improvements are.

2) Procedure: We conducted stability analysis only for those LLMs that consistently preserve the original test semantics of the automatically generated test suites. For those models, and for each generated test suite, we executed our readability improvement approach five times (with the temperature set to 0) to account for non-determinism. For each test, we computed the cosine similarity by considering all the unique permutations of the test and its five versions. We embedded each test using OpenAI text embeddings [12], as they are known to work well with code as input. In particular, we used the `text-embedding-3-small` model.

E. RQ$_3$ (human study)

1) Procedure and Metrics: For each class selected for the readability study (see Column 4 of Table I), we executed both the developer-written test suite and the test suite generated by `Evosuite`, and collected the `Jacoco` reports. We ranked the tests generated by `Evosuite` for a given class by length in ascending order (to give preference to shorter over longer tests), and, for each test, we looked for developer-written tests with the same coverage profile (i.e., statements, methods, and branches) of the class under test as the automatically generated tests under analysis. If no perfect match existed, we selected the first automatically generated test with the *closest match* to a developer-written test (i.e., the one with the minimum coverage differences). We followed this procedure to ensure a comparable semantics (approximated as the coverage semantics) of the automatically generated tests and developer-written tests being considered in the human study. At the end of the selection, we have two tests per class, i.e., the automatically generated test, and the developer-written test that best matches the coverage profile of the former. On average, the automatically generated tests have a length of 5.3 statements, which is a reasonable size for test cases that need to be manually evaluated. We then executed our readability improvement approach five times, once for each of the five LLMs that showed semantic preservation (RQ$_1$). Hence, in total for each class we have six tests, among which five that are automatically generated and improved, and one developer-written, giving a total of 60 tests, as we have two classes per project and five projects.

We designed the survey for the human study to be at most 30-minutes long, to avoid that fatigue would affect the readability assessment of the tests. As we have two classes per project, we split the survey in two parts, i.e., a group of developers would evaluate the tests related to the first class of each project, while the other group of developers would evaluate the tests related to the second class of each project. In this way, a single developer is asked to evaluate 30 tests in 30 minutes, i.e., one test per minute, which was deemed feasible in preliminary trials.

We organized each part of the survey into five blocks of questions, i.e., one per project, where in each block we placed the six tests related to the class under test for the specific project. We then asked developers to assign a score to each of the test, from -2 (the test code is very unreadable) to 2 (the test code is very readable). In each block of questions, we also added an optional text-box where developers could justify and comment their scores. To avoid a learning effect, we randomized the order in which the tests are presented in the survey (printouts of the survey are in our replication package [53]).

To conduct the study, we hired ten professional developers, i.e., five for the first survey and five for the second one, on *Upwork*[13]. We selected this platform because it is a global freelancing platform that facilitates remote collaboration between clients and professionals, and because it was used in other studies in software engineering [54]. We posted a fixed-priced job with a payment of 20 USD [14]. We calculated the price as 30 USD per hour, based on the average salary of a software developer in the US[15], plus 5 USD for completing the qualification task. We clarified explicitly in the payment terms of the job post that payment would be due only if the qualification task and the survey were successfully completed.

The qualification task consists of finding a functional bug we seeded in a Java class selected from the `CodeDefenders` benchmark [55]. In particular, we selected the `Lift` class among the available 12 classes, as its functionalities are relatively easy to understand in a short amount of time, which is compatible with a qualification test (the source code of the `Lift` class is in our replication package [53]). In total, we interviewed 11 developers, of which one did not pass the qualification test. All the others were able to successfully write a test that exposes the functional bug.

Based on the information we collected during the survey, the ten developers we hired had different experiences in software development, ranging from 1-3 years (1 developer) to more than 5 years (7 developers). Most of them (6 developers) reported to be at an intermediate level regarding their software testing skills, with 3 being expert and one beginner.

F. Results

1) RQ$_1$ (semantic preservation): Columns 1–9 of Table II show the average success rates, in terms of semantic preservation, of the nine evaluated LLMs across five repetitions, per project for all the selected tests in the

[12] OpenAI Text Embeddings (September 2024)

[13] https://www.upwork.com (September 2024)

[14] The client assumes the costs of the platform, which consists of a flat-fee per contract initiation and VAT.

[15] Payscale SD hourly rate (September 2024)

	RQ$_1$(semantic preservation)									**RQ$_2$ (stability)**				
	Success Rate (%)									**Cosine Similarity**				
	`gpt-3.5-turbo`	`gpt-4`	`gemini-1.5-pro`	`llama-3-8b`	`llama-3-70b`	`claude-3-haiku`	`claude-3-sonnet`	`mistral-7b`	`mistral-8x7b`	`gpt-3.5-turbo`	`gpt-4`	`gemini-1.5-pro`	`claude-3-haiku`	`claude-3-sonnet`
Cli	100.00	100.00	100.00	0.00	50.00	100.00	100.00	0.00	20.00	0.87	0.95	0.94	0.90	0.94
Csv	100.00	100.00	100.00	20.00	60.00	100.00	100.00	0.00	40.00	0.83	0.82	0.92	0.88	0.89
Lang	100.00	100.00	100.00	40.00	0.00	100.00	100.00	0.00	10.00	0.96	0.98	0.94	0.88	0.88
Gson	100.00	100.00	100.00	0.00	80.00	100.00	100.00	0.00	30.00	0.97	0.99	0.88	0.89	0.94
Chart	100.00	100.00	100.00	40.00	60.00	100.00	100.00	0.00	0.00	0.96	0.95	0.88	0.81	0.87
Avg	100.00	100.00	100.00	20.00	50.00	100.00	100.00	0.00	20.00	0.92	0.94	0.91	0.87	0.90

TABLE II: Results for RQ$_1$ and RQ$_2$. The table shows the average success rate (Columns 1–9) and the average cosine similarity (Columns 10–14) of each LLM across five repetitions, for each project, considering all the test cases and classes.

respective classes. Results show that different models exhibit significant differences in their ability to preserve the test semantics. By looking at the *Avg* row, the `gpt-3.5-turbo`, `gpt-4`, `gemini-1.5-pro`, `claude-3-haiku`, and `claude-3-sonnet` models achieved an average success rate across all projects of 100%, highlighting their reliability and suitability for the readability task. In contrast, the `mistral-7b`, `llama-3-8b`, `mistral-8x7b`, and `llama-3-70b` models showed significantly lower success rates, with average success rates considering all projects of 0%, 20%, 20%, and 50%, respectively, indicating a lower ability to preserve the test semantics when improving their readability. Among models that are not always semantically-preserving, bigger models tend to be better than their smaller counterparts (i.e., `mistral-8x7b` has a higher success rate than `mistral-7b`, and `llama-3-70b` is better than `llama-3-8b`). Most of the times the readability improvements of such LLMs are not semantically-preserving due to compilation errors. By inspecting the tests improved by such LLMs, we noticed that the most frequent semantic-breaking change is the modification of the name of the class when instantiating the constructor. Another common semantic-breaking change is the name of the class methods called in the test.

RQ$_1$ (semantic preservation): *Overall, most considered LLMs (5 out of 9) are able to follow the instructions in the prompt and do not change the semantics of the tests they are improving.*

2) RQ$_2$ (stability): Columns 10–14 of Table II show the average cosine similiarity values for each LLM across five repetitions, per project, considering all the test cases and classes. All the models seem to produce very similar tests across repetitions, as the average cosine similarity across projects ranges from a minimum of 0.87 (`claude-3-haiku`), to a maximum of 0.94 (`gpt-4`).

Figure 2 shows the distribution of cosine similarities for each LLM and project. The x-axis shows the five projects, while the y-axis shows the cosine similarity values. We observe that the `gpt-4` and `gpt-3.5-turbo` LLMs show generally narrow distributions for the Lang, Gson and Chart projects. On the contrary, `gemini-1.5-pro` seems to be more stable in the remaining projects, i.e., Cli and Csv, while its results vary more on Lang, Gson and Chart. The `claude-3-haiku` and `claude-3-sonnet` LLMs show similar patterns across projects. Considering the variability per project, almost all LLMs seem to be quite stable in Cli and Gson classes (except `gpt-3.5-turbo` and `gemini-1.5-pro` respectively), while Csv and Lang classes are those where LLMs exhibit higher variability.

RQ$_2$ (stability): *Overall, all LLMs produce tests with improved readability that have a high similarity between each other, hence exhibiting high stability.*

3) RQ$_3$ (human study): Figure 3 shows the distributions of readability scores given by developers for each LLM and across all the classes for all the projects. The x-axis shows the models we considered for the study, as well as the developer-written tests (`dev-written`). The y-axis shows the readability score ranging from -2 to 2. Overall, we observe that the boxplots for all the models, including the `dev-written` tests, overlap for the majority of their respective interquantile ranges. Although the medians of the `gpt-3.5-turbo` and `claude-3-sonnet` models seem to be higher than the other models (including the `dev-written` tests), the respective differences with the `dev-written` tests are not statistically significant according to the Wilcoxon test [56] (with a significance level at $\alpha < 0.05$). Due to the small sample size (50 = 10 developers × 5 projects), the statistical power β did not reach the conventional threshold of 0.8, commonly used

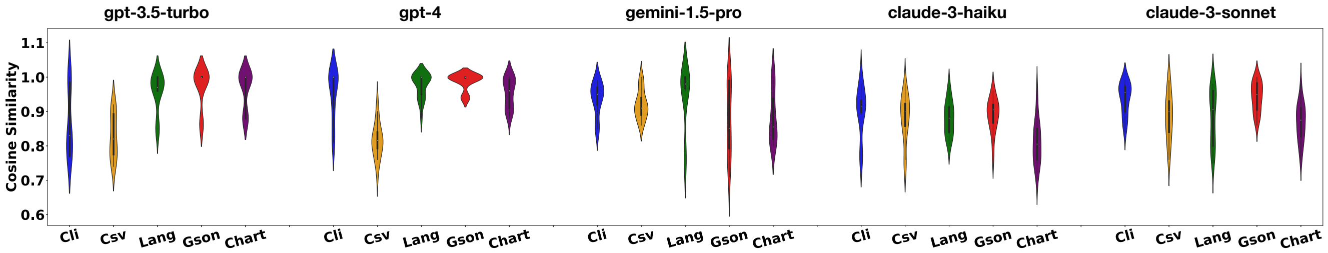

Fig. 2: Results for RQ$_2$. The figure shows the distributions of cosine similarity values for each LLM, for each project, considering all the test cases and classes. The x-axis shows the different projects for each LLM, while the y-axis shows the cosine similarity, which ranges from -1 to 1.

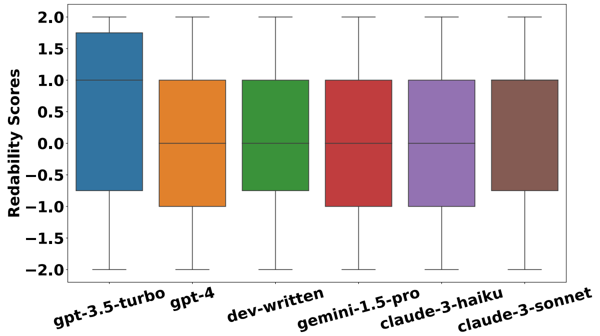

Fig. 3: Distributions of readability scores for each LLM across projects (RQ$_3$). The x-axis shows the different LLM models, plus the tests written by developers (`dev-written`); the y-axis shows the readability scores, ranging from -2 (the test code is very unreadable) to 2 (the test code is very readable).

to accept with confidence the null hypothesis of no significant difference. However, the boxplots show a clear picture, with substantial readability score overlap and no indication of lower readability w.r.t. developer-written tests.

> **RQ$_3$ (human study)**: *Overall, developers judge the readability of developer-written tests similar to that of LLM-improved tests.*

G. Threats to validity

Internal validity. Internal validity threats might come from the way the empirical study was carried out. To ensure a fair comparison between LLM-improved tests and developer-written tests, we selected tests with a comparable coverage. Moreover, we selected known and documented classes with high-quality test suites, to avoid comparing LLM-improved tests with developer-written tests that have a low readability. Such classes were chosen with a systematic procedure, by first adopting automated filters and then resorting to an internal survey with PhD students, to manually assess the suitability of those classes for the readability task. Another internal validity threat comes from the design of the prompt. We mitigated this threat by following existing guidelines for prompt engineering. Other internal threats come from the way we designed the survey for the human study. We mitigated these threats, by (1) splitting the survey into two parts, to avoid a fatigue effect that would result from the assessment of a high number of tests; (2) randomizing the order in which tests are presented to developers during the survey, to avoid the learning effect.

External validity. External validity threats concern the generalizability of our results. We mitigated these threats by selecting five well-known Java projects, and nine LLMs from different providers. Results could also be affected by the population of developers registered on Upwork. We mitigated this threat by designing a qualification test to assess the skills of developers before completing the survey.

Conclusion validity. Threats to conclusion validity may come from random variations in the experiments. We mitigated this threat by studying semantic preservation and stability using five repetitions of our readability improvement approach, as LLMs are notoriously non-deterministic [49].

Construct validity. Construct validity threats are related to the choice of inappropriate metrics. To measure test semantic preservation we used the coverage information provided by `Jacoco` during the execution. Although tracking the whole computation state would have been more precise, coverage is a computationally efficient proxy for semantics, and it is widely used in the literature [3].

VI. Conclusion and Future Work

Search-based unit test generators have shown to be very effective at exercising the functionalities of software classes. However the generated tests have low readability, as the test names do not reflect the semantics of the execution, and identifiers are generic and hard to interpret. Large language models (LLMs), on the other hand, generate readable tests, but with an overall lower effectiveness. In this paper, we proposed to combine the effectiveness of search-based unit test generators with the ability of LLMs to improve and manipulate natural language and source code. Our approach takes as input a class under test, generates a test suite, and uses an LLM to improve each test by focusing on the test names and identifiers. Results show that LLM-improved tests maintain the semantics of the original tests, while improving their readability, making it comparable to that of developer-written tests. In our future work, we plan to extend our approach with Retrieval Augmented Generation [57], to take into account additional knowledge when the LLM is prompted for readability improvements.

VII. Data Availability

Our replication package is publicly available [53], making our results reproducible.

References

[1] M. Harman, "Search based software engineering," in *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, ser. Lecture Notes in Computer Science, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds., vol. 3994. Springer, 2006, pp. 740–747. [Online]. Available: https://doi.org/10.1007/11758549_100

[2] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, G. S. Avrunin and G. Rothermel, Eds. ACM, 2004, pp. 119–128. [Online]. Available: https://doi.org/10.1145/1007512.1007528

[3] G. Fraser and A. Arcuri, "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*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[4] ——, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.14

[5] A. Arcuri, "Many independent objective (MIO) algorithm for test suite generation," in *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings*, ser. Lecture Notes in Computer Science, T. Menzies and J. Petke, Eds., vol. 10452. Springer, 2017, pp. 3–17. [Online]. Available: https://doi.org/10.1007/978-3-319-66299-2_1

[6] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Software Eng.*, vol. 44, no. 2, pp. 122–158, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2663435

[7] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 2022, pp. 168–172. [Online]. Available: https://doi.org/10.1145/3510454.3516829

[8] M. Olsthoorn, D. Stallenberg, and A. Panichella, "Syntest-javascript: Automated unit-level test case generation for javascript," in *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, ser. SBFT '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 21–24. [Online]. Available: https://doi.org/10.1145/3643659.3643928

[9] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 394–397. [Online]. Available: https://doi.org/10.1109/ICST.2018.00046

[10] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings*, ser. Lecture Notes in Computer Science, T. Menzies and J. Petke, Eds., vol. 10452. Springer, 2017, pp. 18–32. [Online]. Available: https://doi.org/10.1007/978-3-319-66299-2_2

[11] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[12] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 107–118. [Online]. Available: https://doi.org/10.1145/2786805.2786838

[13] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: would you name your children thing1 and thing2?" in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds. ACM, 2017, pp. 57–67. [Online]. Available: https://doi.org/10.1145/3092703.3092727

[14] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[16] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Trans. Software Eng.*, vol. 50, no. 1, pp. 85–105, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2023.3334955

[17] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 572–576. [Online]. Available: https://doi.org/10.1145/3663529.3663801

[18] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1703–1726, 2024. [Online]. Available: https://doi.org/10.1145/3660783

[19] A. Sapozhnikov, M. Olsthoorn, A. Panichella, V. Kovalenko, and P. Derakhshanfar, "Testspark: Intellij idea's ultimate test generation companion," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 30–34. [Online]. Available: https://doi.org/10.1145/3639478.3640024

[20] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs SBST: A comparative assessment of unit test suite generation," *IEEE Trans. Software Eng.*, vol. 50, no. 6, pp. 1340–1359, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3382365

[21] M. L. Siddiqa, J. C. Santos, R. H. Tanvirb, N. Ulfatc, F. Al Rifatd, and V. C. Lopes, "An empirical study of using large language models for unit test generation," *arXiv preprint arXiv:2305.00418*, 2023.

[22] Z. Xie, Y. Chen, C. Zhi, S. Deng, and J. Yin, "Chatunitest: a chatgpt-based automated unit test generation tool," *CoRR*, vol. abs/2305.04764, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.04764

[23] G. Gay, "Generating effective test suites by combining coverage criteria," in *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings*, ser. Lecture Notes in Computer Science, T. Menzies and J. Petke, Eds., vol. 10452. Springer, 2017, pp. 65–82. [Online]. Available: https://doi.org/10.1007/978-3-319-66299-2_5

[24] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, M. d'Amorim, Ed. ACM, 2024, pp. 185–196. [Online]. Available: https://doi.org/10.1145/3663529.3663839

[25] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 157–173, 2024. [Online]. Available: https://doi.org/10.1162/tacl_a_00638

[26] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Information and Software Technology*, vol. 104, pp. 236–256, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584917304950

[27] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verification Reliab.*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: https://doi.org/10.1002/stvr.294

[28] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, “A systematic review of the application and empirical investigation of search-based test case generation,” *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 742–762, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.52

[29] C. Pacheco and M. D. Ernst, “Randoop: feedback-directed random testing for java,” in *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902

[30] G. Jahangirova and V. Terragni, “SBFT tool competition 2023 - java test case generation track,” in *IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT@ICSE 2023, Melbourne, Australia, May 14, 2023*. IEEE, 2023, pp. 61–64. [Online]. Available: https://doi.org/10.1109/SBFT59156.2023.00025

[31] A. Arcuri, “Restful API automated test case generation with evomaster,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 3:1–3:37, 2019. [Online]. Available: https://doi.org/10.1145/3293455

[32] S. Lukasczyk, F. Kroiß, and G. Fraser, “An empirical study of automated unit test generation for python,” *Empir. Softw. Eng.*, vol. 28, no. 2, p. 36, 2023. [Online]. Available: https://doi.org/10.1007/s10664-022-10248-w

[33] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software testing with large language models: Survey, landscape, and vision,” *IEEE Trans. Software Eng.*, vol. 50, no. 4, pp. 911–936, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3368208

[34] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, “A3test: Assertion-augmented automated test case generation,” *Inf. Softw. Technol.*, vol. 176, p. 107565, 2024. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107565

[35] L. Plein, W. C. Ouédraogo, J. Klein, and T. F. Bissyandé, “Automatic generation of test cases based on bug reports: a feasibility study with large language models,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 360–361. [Online]. Available: https://doi.org/10.1145/3639478.3643119

[36] T. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[37] G. A. Campbell, “Cognitive complexity: an overview and evaluation,” in *Proceedings of the 2018 International Conference on Technical Debt, TechDebt@ICSE 2018, Gothenburg, Sweden, May 27-28, 2018*, R. L. Nord, F. Buschmann, and P. Kruchten, Eds. ACM, 2018, pp. 57–58. [Online]. Available: https://doi.org/10.1145/3194164.3194186

[38] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models,” in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 919–931. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00085

[39] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, “Effective test generation using pre-trained large language models and mutation testing,” *Inf. Softw. Technol.*, vol. 171, p. 107468, 2024. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107468

[40] R. P. Buse and W. R. Weimer, “Learning a metric for code readability,” *IEEE Transactions on software engineering*, vol. 36, no. 4, pp. 546–558, 2009.

[41] M. M. Barón, M. Wyrich, and S. Wagner, “An empirical validation of cognitive complexity as a measure of source code understandability,” in *ESEM ’20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*, M. T. Baldassarre, F. Lanubile, M. Kalinowski, and F. Sarro, Eds. ACM, 2020, pp. 5:1–5:12. [Online]. Available: https://doi.org/10.1145/3382494.3410636

[42] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, “The effect of modularization and comments on program comprehension,” in *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, S. Jeffrey and L. G. Stucki, Eds. IEEE Computer Society, 1981, pp. 215–223. [Online]. Available: http://dl.acm.org/citation.cfm?id=802534

[43] J. C. Hofmeister, J. Siegmund, and D. V. Holt, “Shorter identifier names take longer to comprehend,” *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 417–443, 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9621-x

[44] T. Sedano, “Code readability testing, an empirical study,” in *29th IEEE International Conference on Software Engineering Education and Training, CSEET 2016, Dallas, TX, USA, April 5-6, 2016*. IEEE, 2016, pp. 111–117. [Online]. Available: https://doi.org/10.1109/CSEET.2016.36

[45] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, “The impact of test case summaries on bug fixing performance: an empirical investigation,” in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 547–558. [Online]. Available: https://doi.org/10.1145/2884781.2884847

[46] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, “Deeptc-enhancer: Improving the readability of automatically generated tests,” in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 287–298. [Online]. Available: https://doi.org/10.1145/3324884.3416622

[47] P. Delgado-Pérez, A. Ramírez, K. J. Valle-Gómez, I. Medina-Bulo, and J. R. Romero, “Interevo-tr: Interactive evolutionary test generation with readability assessment,” *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2580–2596, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3227418

[48] G. Gay, “Improving the readability of generated tests using GPT-4 and chatgpt code interpreter,” in *Search-Based Software Engineering - 15th International Symposium, SSBSE 2023, San Francisco, CA, USA, December 8, 2023, Proceedings*, ser. Lecture Notes in Computer Science, P. Arcaini, T. Yue, and E. M. Fredericks, Eds., vol. 14415. Springer, 2023, pp. 140–146. [Online]. Available: https://doi.org/10.1007/978-3-031-48796-5_11

[49] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, “LLM is like a box of chocolates: the non-determinism of chatgpt in code generation,” *CoRR*, vol. abs/2308.02828, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.02828

[50] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, “A systematic survey of prompt engineering in large language models: Techniques and applications,” *CoRR*, vol. abs/2402.07927, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.07927

[51] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, “Chain-of-thought prompting elicits reasoning in large language models,” in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[52] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[53] Anonymous, 2024. [Online]. Available: https://anonymous.4open.science/r/readability-llms-3CA6/README.md

[54] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, “An empirical validation of oracle improvement,” *IEEE Trans. Software Eng.*, vol. 47, no. 8, pp. 1708–1728, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2934409

[55] G. Fraser, A. Gambi, M. Kreis, and J. M. Rojas, “Gamifying a software testing course with code defenders,” in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, E. K. Hawthorne, M. A. Pérez-Quiñones, S. Heckman, and J. Zhang, Eds. ACM, 2019, pp. 571–577. [Online]. Available: https://doi.org/10.1145/3287324.3287471

[56] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: http://www.jstor.org/stable/3001968

[57] P. S. H. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for

knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html