Using Large Language Models to Generate JUnit Tests: An Empirical Study

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Ridwanul Hasan Tanvir
rpt5409@psu.edu
Pennsylvania State University
University Park, PA, USA

Noshin Ulfat
noshin.ulfat@iqvia.com
IQVIA Inc.
Dhaka, Bangladesh

Fahmid Al Rifat
fahmid@cse.uiu.ac.bd
United International University
Dhaka, Bangladesh

Vinícius Carvalho Lopes
vlopes@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT

A code generation model generates code by taking a prompt from a code comment, existing code, or a combination of both. Although code generation models (*e.g.*, GitHub Copilot) are increasingly being adopted in practice, it is unclear whether they can successfully be used for unit test generation without fine-tuning for a strongly typed language like Java. To fill this gap, we investigated how well three models (Codex, GPT-3.5-Turbo, and StarCoder) can generate unit tests. We used two benchmarks (HumanEval and Evosuite SF110) to investigate the effect of context generation on the unit test generation process. We evaluated the models based on compilation rates, test correctness, test coverage, and test smells. We found that the Codex model achieved above 80% coverage for the HumanEval dataset, but no model had more than 2% coverage for the EvoSuite SF110 benchmark. The generated tests also suffered from test smells, such as Duplicated Asserts and Empty Tests.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Instance-based learning*.

KEYWORDS

test generation, unit testing, large language models, test smells, junit

ACM Reference Format:

Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In *Proceedings of The 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EASE 2024, 18–21 June, 2024, Salerno, Italy
© 2024 ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Unit testing [8] is a software engineering activity in which individual units of code are tested in isolation. This is an important activity because it helps developers identify and fix defects early on in the development process and understand how the various units of code in a software system fit together and work as a cohesive whole. Despite its importance, in practice, developers face difficulties when writing unit tests [17, 36, 55, 65]. This leads to a negative effect: developers may not write tests for their code. In fact, a prior study [26] showed that out of 82,447 studied GitHub projects, only **17%** of them contained test files.

Since implementing test cases to achieve good code coverage is a time-consuming and error-prone task, prior works [60, 66] developed techniques to automatically generate unit tests. Although automatically generated unit tests help increase code coverage [6, 58], they are still not frequently used in practice [23].

With the advances of large language models (LLMs), LLM-based code generation tools (*e.g.*, GitHub Copilot) are increasingly becoming part of day-to-day software development. A survey of 500 US-based developers showed that 92% of them are using LLM-based coding assistants both for work and personal use [61]. Part of this fast widespread adoption is that LLMs automate repetitive tasks so that they can focus on higher-level, challenging tasks [71]. With the increasing popularity of code generation LLMs, prior works investigated the correctness of the generated code [18], their quality [62], security [50] and whether they can be used for API learning tasks [31], and code complexity prediction [63]. However, it is currently unclear the effectiveness of using prompt-based pre-trained code generation models to generate *unit tests* for strongly typed languages such as Java. In fact, prior works [4, 14] have shown that LLMs perform better for weakly typed languages (*e.g.*, Python and JavaScript) but not as well for strongly typed languages. This is

partially due to the limited training sets availability and the fact that strongly typed languages have strict type-checking that can prevent the code from even compiling.

In light of this research gap, we conducted an empirical study using three LLMs (Codex [16], GPT-3.5-Turbo [1] and StarCoder [39]) to generate JUnit5 tests for classes in the HumanEval dataset's Java version [5] and 47 open-source projects from the SF110 dataset [20]. In our study, we investigate how well LLMs can generate JUnit tests (RQ1) and how different context styles (*e.g.*, only using the method under test, the presence/ absence of JavaDocs *etc.*) provided as input to an LLM can influence its performance (RQ2). We examined the generated tests with respect to their *compilation rates*, *correctness*, *code coverage*, and quality (in terms of test smells). While concurrent works [38, 57] studied the usefulness of LLM as a *helper* for search-based test generation techniques on weakly typed languages (*i.e.*, Python and JavaScript), our work investigates whether LLMs can be used off-the-shelf to generated unit tests for a strongly-typed language like Java. Moreover, we examine these generated tests in terms not only of their *correctness*, but also their *quality*, as well as the effectiveness of different *context styles*.

The **contributions** of our work are: ① A systematic study of three LLMs for zero-shot unit test generation for 194 classes from 47 open-source projects in the SF110 dataset [21] and 160 classes from the HumanEval dataset [5]. ② An investigation of the quality of the produced unit tests by studying the prevalence of test smells in the generated unit tests. ③ A comparison of how different context styles affect the performance of LLMs in generating tests. ④ A discussion about the implication of using code generation models for unit test generation in a Test Driven Development (TDD) environment. All the scripts used to gather the data and spreadsheets compiling all the results are available on Zenodo [1].

2 BACKGROUND

2.1 Unit Tests & Test Smells

The goal of ***unit testing*** is to validate that each program unit is working as intended and meets its requirements [37]. A ***unit*** refers to a piece of code that can be isolated and examined independently (*e.g.*, functions/methods, classes, *etc.*). Just like production code, unit tests need to be not only *correct* but also satisfy other quality attributes, such as *maintainability* and *readability* [26].

Unit test smells (henceforth "test smells") are indicators of potential problems, inefficiencies, or bad programming/design practices in a unit test suite [28, 29, 49, 51, 67]. There are many test smell types, ranging from tests that are too slow/fragile to tests that are too complex or too tightly coupled to implementing the code under test [45]. For example, the Java code in Listing 1 has a unit test for a method from the `LargestDivisor` class. It checks whether the Method Under Test (MUT) returns the largest divisor of a number. Although this test is correct, there is no explanation for the expected outputs passed to the assertions, which is a case of the Magic Number Test smell [45]. It also has multiple assertions in the same test method, an example of Assertion Roulette smell [67].

[1]https://doi.org/10.5281/zenodo.10530787

LargestDivisorTest.java

```
public class LargestDivisorTest {
    @Test
    void testLargestDivisor() {
        assertEquals(5, LargestDivisor.largestDivisor(15));
        assertEquals(1, LargestDivisor.largestDivisor(3));
    }
}
```

Listing 1: Example of Unit Test and Unit Test Smell

2.2 Code Generation

Large Language Models (LLMs) are advanced AI systems capable of understanding and generating human-like text. They can bed used to answer questions, create content, and even engage in conversation. ***Code LLMs*** (henceforth simply "LLMs") are a specialized type of LLMs trained on source code to help with code-related tasks, *e.g.*, code completion [34, 35, 64], search [19], and summarization [25]. They are designed to generate source code from a given ***prompt*** [3], such as a text written in natural language, pseudocode, code comments *etc.* These techniques may also take into account the surrounding ***context*** when generating the code, such as file/variable names, other files in the software system, *etc.*

3 METHODOLOGY

In this work, we answer two research questions.

RQ1 ***How well can LLMs generate JUnit tests?***

We used GPT-3.5-Turbo, StarCoder, and Codex to generate unit tests for competitive programming assignments from the Java version of the HumanEval dataset [5] as well as 47 open-source projects from the EvoSuite SF110 benchmark dataset[20]. We measured the LLMs' performance by computing the test's branch/line coverage, correctness, and quality (in terms of test smells). We also compared the performance of these models with Evosuite [20], an existing state-of-the-art approach.

RQ2 ***How do different code elements in a context influence the performance of LLMs in generating JUnit tests?***

When developers use LLMs to generate JUnit tests, they create a ***prompt*** (*e.g.*, *"Write a JUnit test to verify that* `login(req)` *returns ...*") and the method (unit) under test becomes the ***context*** for that prompt. Since the unit under test (context) can include several *code elements*, we investigate how these different elements affect the generated tests. To answer RQ2, we conducted a controlled experiment in which we created 3 different scenarios for the HumanEval [5, 16], and 4 scenarios for 47 open-source projects from the EvoSuite SF110 dataset[20]. Each scenario contains a different set of code elements. Then, we use Codex, GPT-3.5-Turbo, and StarCoder to generate JUnit tests for each scenario. We measured their performance in terms of compilation rates, code coverage, the total number of correct unit tests, and the incidence of test smells.

3.1 Answering RQ1

We followed a three-step systematic process to investigate how well LLMs can generate unit tests: ① we collected **160** Java classes from the **multilingual HumanEval dataset** [5] and **194** Java classes from **47** projects in the **Evosuite SF110 benchmark dataset** [13, 21]; ② we generated JUnit5 tests using three LLMs; ③ we computed the compilation rates, correctness, number of smells, as well as the

line/branch coverage for the generated tests and compared with Evosuite v1.2.0, which is a state-of-the-art unit test generation tool [20]. In this paper, ***methods*** are our units under test.

3.1.1 Data Collection. We use the multilingual HumanEval dataset [5] because it has been widely used in prior works [24, 47, 62] to evaluate code LLMs. Similarly, we use the SF110 dataset because it is a popular benchmark for unit test generation [22].

```
GreatestCommonDivisor.java
1  class GreatestCommonDivisor {
2      /**
3       * Return the greatest common divisor of two integers a and b.
4       * > greatestCommonDivisor(3, 5)
5       * 1
6       */
7      public static int greatestCommonDivisor(int a, int b) {
8          if (a == 0) return b;
9          return greatestCommonDivisor(b % a, a);
10     }
11 }
```

Listing 2: Sample from the extended HumanEval [5]

– The **multilingual HumanEval dataset** [5] contains **160** *prompts* describing programming problems for Java and other programming languages crafted from the original Python-based HumanEval [16]. However, this multilingual version does not provide a solution for each prompt. Thus, we wrote the solution for each problem and tested our implementation using the provided test cases. Listing 2 shows a sample taken from this dataset, where the prompt is in lines 1–7 and the solution is in lines 8–11.

– The **SF110 dataset**, which is an Evosuite benchmark consisting of 111 open-source Java projects retrieved from SourceForge. This benchmark contains 23,886 classes, over 800,000 bytecode-level branches, and 6.6 million lines of code [22].

Class and Method Under Test Selection. Each class in the multilingual HumanEval [5] has one `public static` method and may also contain private "helper" methods to aid the solution implementation. In this study, ***all*** the public static methods are selected as methods under test (**MUTs**).

For the SF110 benchmark, we first retrieved only the classes that are `public` and ***not*** `abstract`. We then discarded test classes (*i.e.*, placed on a src/test folder, or that contains the keyword "Test" in its name). Next, we identified *testable methods* by applying *inclusion* and *exclusion* criteria. The *exclusion* criteria are applied to the ***non-static*** methods that **(E1)** have a name starting with "get" and takes no parameters, ***or*** **(E2)** have a name starting with "is", takes no parameter and returns a `boolean` value, ***or*** **(E3)** have a name starting with "set", ***or*** **(E4)** override the ones from `java.lang.Object` (*i.e.*, `toString()`, `hashCode()`, *etc.*). The exclusion criteria **E1–E3** are meant to disregard "getter" and "setter" methods. The inclusion criteria are that the method has **(I1)** a `public` visibility, **(I2)** a `return` value, ***and*** **(I3)** a *good* JavaDoc. A *good* JavaDoc is one that **(i)** has a description *or* has a non-empty `@return` tag, and **(ii)** all the method's parameters have an associated description with `@param` tag. After this step, we obtained a total of 30,916 methods under test (MUTs) from 2,951 classes. Subsequently, we disregard projects based on the number of retrieved testable methods (MUTs). We kept projects with at least one testable method (*i.e.*, first quartile) and at most 31 testable methods (*i.e.*, third quartile), obtaining a total of 53 projects. This filtering aimed to remove projects with too *little* or too *many* MUTs, which would exceed the limit of the number of tokens that the models can generate. We then removed 6 of these projects in which we could not compile their source code, obtaining **47** projects and a total of **411** MUTs from **194** classes.

3.1.2 Unit Test Generation. We used Codex, GPT-3.5-Turbo, and StarCoder to generate JUnit tests. **Codex** is a 12 billion parameters LLM [16] descendant of the GPT-3 model [11] which powers GitHub Copilot. In this study, we used code-davinci-002, the most powerful codex model version of Codex. **GPT-3.5-turbo** is the model that powers the ChatGPT chatbot. It allows multi-turn conversation, and it can be instructed to generate code [1]. **StarCoder** is a 15.5 billion parameter open-source code generation model with 8,000 context length and has infilling capabilities. In this work, we used the base model from the StarCoder code LLM series.

To generate the JUnit tests, we performed a two-step process:

① ***Context and Prompt Creation***: We created a ***unit test prompt*** (henceforth "prompt"), which instructs the LLM to generate **10** test cases for a specific method, and a ***context***, which encompasses the whole code from the method's declaring class as well as import statements to core elements from the JUnit5 API. Listing 3 illustrates the structure of a prompt and context, in which lines 1-9 and lines 10-20 are part of the *context* and *prompt*, respectively. The context starts with a comment indicating the class' file name followed by its full code (*i.e.*, its package declaration, imports, fields, methods, *etc.*). Similarly, the prompt starts with a comment indicating the expected file name of the generated unit test. Since a class can have more than one testable method, we generated one unit test file for each testable method in a class and appended a suffix to avoid duplicated test file names. A suffix is a number that starts from zero. After this code comment, the prompt includes the same package declaration and import statements from the class. It also has import statements to the `@Test` annotation and the `assert*` methods (*e.g.*, `assertTrue(...)`) from JUnit5. Subsequently, the prompt contains the test class' JavaDoc that specifies the MUT, and how many test cases to generate. The prompt ends with the test class declaration followed by a new line (`\n`), which will trigger the LLM to generate code to complete the test class declaration.

```
classNameSuffixTest.java
1  // ${className}.java
2  ${packageDeclaration}
3  ${importedPackages}
4  class ${className}{
5    /* ... code before the method under test  ... */
6    public ${methodSignature}{ /* ... method implementation ... */ }
7    /* ... code after the method under test  ... */
8  }
9
10 // ${className}${suffix}Test.java
11 ${packageDeclaration}
12 ${importedPackages}
13 import org.junit.jupiter.api.Test;
14 import static org.junit.jupiter.api.Assertions.*;
15
16 /**
17 * Test class of {@link ${className}}.
18 * It contains ${numberTests} unit test cases for the
19 * {@link ${className}#${methodSignature}} method.
20 */
21 class ${className}${suffix}Test {
```

Listing 3: Prompt template for RQ1

② ***Test Generation***: Although all used LLMs can generate code, they have technical differences in terms of number of tokens they

can handle. Thus, we took slightly different steps to generate tests with these LLMs. We used the OpenAI API to generate tests using the **Codex** model. Codex can take up to 8,000 tokens as input and generate up to 4,000 tokens. Thus, we configured this model in two ways: one to generate up to 2,000 tokens and another to generate up to 4,000 tokens. We will call each of them **Codex (2K)** and **Codex (4K)**, respectively. For both cases, we set the model's `temperature` as zero in order to produce more deterministic and reproducible output motivated by previous studies [15, 53, 56]. The rest of its inference parameters are set to their default values.

GPT-3.5-Turbo is also accessible via the OpenAI API. It can take up to 4,096 tokens as input and generate up to 2,048 tokens. We asked this LLM to generate up to 2,000 tokens and dedicated the rest (2,096) to be used as input. Its temperature is also set to zero and the other parameters are set to their defaults. Moreover, we set the system role's content to *"You are a coding assistant. You generate only source code."* and the user role's content to the context and prompt. Then, the assistant role outputs the generated test. For **StarCoder**, we used the StarCoderBase model available on HuggingFace library[2]. It has an 8,000 tokens context window combining the input prompt tokens and the output tokens. We limit the output token to 2,000 tokens to align the experiment with the other two models. We also keep the same inference parameters as the Codex model.

3.1.3 Data Analysis and Evaluation. We compiled all the unit tests together with their respective production code and required libraries. As we compiled the code and obtained compilation errors, we observed that several of these errors were caused by simple *syntax* problems that could be automatically fixed through *heuristics*. Specifically, we noticed that LLMs may ***(i)*** generate an *extra* test class that is incomplete, ***(ii)*** include natural language explanations before and/or after the code, ***(iii)*** repeat the class under test and/or the prompt, ***(iv)*** change the package declaration or ***(v)*** remove the package declaration, ***(vi)*** generate integer constants higher than `Integer.MAX_VALUE`, ***(vii)*** generate incomplete unit tests after it reaches its token size limit. Thus, we developed **7** heuristics (**H1**–**H7**) to automatically fix these errors :

- **H1** It removes any code found *after* any of the following patterns: `"</code>"`, `"\n\n// {CUT_classname}"`, and `"\n```\n\n##"`.
- **H2** It keeps code snippets within backticks (*i.e.*, ````code```) and removes any text before and after the backticks.
- **H3** It removes the original prompt from the generated unit test.
- **H4** It finds the package declaration in the unit test and renames it to the package of the CUT.
- **H5** It adds the package declaration if it is missing.
- **H6** It replaces large integer constants by `Integer.parseInt(n)`.
- **H7** It fixes incomplete code by iteratively deleting lines (from bottom to top) and adding 1-2 curly brackets. At each iteration, it removes the last line and adds one curly bracket. If the syntax check fails, it adds two curly brackets and checks the syntax again. If it fails, it proceeds to the next iteration by removing the next line (bottom to top). The heuristic stops if the syntax check passes or it finds the class declaration (*i.e.*, "`class ABC`"), whichever condition occurs first.

[2]https://huggingface.co

Metrics. We ran each generated unit test with JaCoCo [2] to compute the ***line coverage***, ***branch coverage*** and ***test correctness*** metrics. ***Branch Coverage*** [33] measures how many branches are covered by a test, *i.e.*, $\frac{Number\ of\ visited\ branches}{Total\ number\ of\ branches} \times 100$. ***Line Coverage*** measures how many lines were executed by the unit test out of the total number of lines [32], *i.e.*, $\frac{Number\ of\ executed\ lines}{Total\ number\ of\ lines} \times 100$. ***Test Correctness*** measures how effectively an LLM generates correct input/output pairs. We assume that the code under test is implemented correctly. The reasoning behind this assumption is twofold: the HumanEval dataset contains common problems with well-known correct solutions, and the SF110 projects are ***mature*** open-source projects. Given this assumption, a failing test case is considered to be *incorrect*. Thus, we compute the number of generated unit tests that did not fail.

We ran the tests using a timeout of **2** and **10** minutes for the HumanEval and the SF110 datasets, respectively, because we observed generated tests with infinite loops. Moreover, we analyzed the quality of the unit test from the perspective of the **test smells**. To this end, we used TsDetect, a state-of-the-art tool that detects 20 test smell types [51, 52]. Due to space constraints, we provide a list of the test smells detectable by TsDetect with their descriptions in our replication package.

3.2 RQ2: Code Elements in a Context

To investigate how different code elements in a context influence the generated unit test, we first created ***three*** scenarios for the HumanEval dataset and ***four*** for the Evosuite Benchmark.

HumanEval Scenarios: Recall that each MUT in this dataset has a JavaDoc describing the method's expected behavior and examples of input-output pairs (see Listing 1). Thus, we created one scenario (**S1**) that does not contain any JavaDoc (*e.g.*, the JavaDoc from lines 2-6 within Listing 2 is removed from the CUT). The second scenario (**S2**) has the JavaDoc but it does not include input/output examples, only the method's behavior description (*e.g.*, Listing 2 will not have lines 4-5). The last scenario (**S3**) does not include the MUT's implementation, only its signature (*e.g.*, Listing 2 will not have lines 8-10). **S1** and **S2** demonstrate the effect of changing JavaDoc elements. Test-Driven Development (TDD) [8] inspires scenario **S3**, where test cases are written before the code implementation.

SF110 Scenarios: Unlike HumanEval, the classes from SF110 do not necessarily include input/output pairs. Thus, we created scenarios slightly different than before. Scenario **S1** removes (i) any code within the class *before* and *after* the method under test as well as (ii) the class' JavaDoc. Scenario **S2** is the same as S1, but *including* the JavaDoc for the method under test. Scenario **S3** is the same as S2, except that there is no method implementation for the MUT (only its signature). Scenario **S4** mimics S3, but it also includes all the fields and the signatures for the other methods/constructors in the MUT's declaring class. Scenarios **S1** and **S2** demonstrate the effect of having or not having code documentation (JavaDoc). **S3** verifies the usefulness of LLMs for TDD whereas **S4** is used to understand how code elements in a class are helpful for test generation.

After creating each of the scenarios above, we generated unit tests using the same models and following the steps outlined in Section 3.1. Then, we used JUnit5, JaCoCo, and TsDetect to measure test coverage, correctness, and quality. Similar to RQ1, we also compared the results to Evosuite [20].

4 RQ1 RESULTS

We analyze the generated tests according to their: **(i)** ***compilation status***; **(ii)** ***correctness***; **(iii)** ***coverage***; and **(iv)** ***quality***.

4.1 Compilation Status

Table 1 reports the percentage of generated unit tests that are compilable ***before*** and ***after*** applying the heuristic-based fixes described in Section 3.1.3. The number of unit tests and test methods for each model and dataset is shown in the last two columns of Table 1. We obtained a total **2,536** test methods (*i.e.*, a method with an `@Test` annotation) scattered across **572** compilable Java test files for HumanEval and **2,022** test methods within **600** test files for SF110. For comparison, we also ran Evosuite [20] (with default configuration parameters) to generate unit tests for each of the MUTs. Moreover, in the case of HumanEval, we manually created a JUnit5 test for each input/output pair provided in each prompt.

HumanEval Results. On the one hand, we found that ***less than half*** of the unit tests generated by Codex (2K), Codex (4K), and GPT-3.5-Turbo are compilable for the classes in HumanEval. On the other hand, **70%** of StarCoder's generated unit tests compiled. Upon applying heuristic-based fixes, the compilation rates have increased an average of **41%**. The biggest increase was observed for the Codex (2K) model; its compilation rate increased from **37.5%** to **100%**. StarCoder was the LLM that the heuristics were the least able to improve; it only increased the compilation rate by **6.9%**.

SF110 Results. For the SF110 dataset, the compilation rates are lower than the ones observed for HumanEval. Between **2.7%** and **12.7%** of the generated unit tests for the SF110 dataset are compilable across all the studied LLMs. StarCoder was the LLM that generated the highest amount of compilable tests (**12.7%**), whereas Codex (2K) and Codex (4K) had the lowest compilation rate (**2.7%** and **3.4%**, respectively). Similar to HumanEval, the heuristic-based fixes were able to increase the compilation rates by **81%**, on average. Codex was the model with the highest increase; the compilation rates increased from less than **5%** to over **99%**. StarCoder was the model that least benefited with our heuristics; its compilation rate increased by only **57.2%**.

Table 1: Compilation status of the generated unit tests

	LLM	% Compilable	% Compilable after fix	#Test Methods	#Test Classes
HumanEval	GPT-3.5-Turbo	43.1%	81.3%	1,117	130
	StarCoder	70.0%	76.9%	948	123
	Codex (2K)	37.5%	100%	697	160
	Codex (4K)	44.4%	99.4%	774	159
	Evosuite	100%	NA	928	160
	Manual	100%	NA	1,303	160
SF110	GPT-3.5-Turbo	9.7%	85.9%	194	87
	StarCoder	12.7%	69.8%	1,663	368
	Codex (2K)	2.7%	74.5%	1,406	222
	Codex (4K)	3.4%	83.5%	1,039	152
	Evosuite	100%	NA	12,362	1,618

Compilation error root causes. The unit tests that were not fixable through heuristics were those that contained *semantic* errors that failed the compilation. To observe the most common root causes of compilation errors, we collected all the compilation errors and clustered them using K-means [42]. We used the silhouette method [54] to find the number of clusters K ($K = 48$). After inspecting these 48 clusters and making manual adjustments to clusters to fix imprecise clustering, we found that the top 3 compilation errors for HumanEval were caused by ***unknown symbols*** (*i.e.*, the compiler cannot find the symbol), ***incompatible conversion from*** `java.util.List<T>` ***to*** `java.util.List<X>`, and ***incompatible conversion from*** `int[]` ***to*** `java.util.List<Integer>`. Unknown symbols accounted for more than **62%** of the compilation errors. Several of these unknown symbols were caused by invoking non-existent methods or instantiating non-existent classes. For example, StarCoder produced several test cases that invoked the method `java.util.List.of(int,int,int,...)`, which does not exist. For the SF110 dataset, the top 3 compilation errors were ***unknown symbols***, ***class is abstract; cannot be instantiated***, and ***no suitable constructor found***.

4.2 Test Correctness

We executed each test that were compilable after our automated fix. We considered a unit test to be ***correct*** if it had a success rate of **100%** (*i.e.*, *all* of its test methods passed) whereas a ***somewhat correct*** unit test is one that had *at least one* passing test method. As explained in Section 3.1.3, the reasoning behind these metrics is that the HumanEval has a canonical solution which is the *correct* implementation for the problem. Thus, a correct test must not fail (or else the input/output generated does not match the benchmark's problem). Similarly, as the SF110 benchmark is a popular benchmark for automatic test generation containing mature open-source projects, they have a higher probability that they are functionally correct. Both metrics are reported in Table 2.

HumanEval Results. StarCoder generated the highest amount of correct unit tests (≈81%). Although GPT-3.5-Turbo only produced **52%** correct unit tests, it was the model that generated the highest amount of tests that have at *at least one* passing test method (**92.3%**). We also found that increasing Codex's token size did not yield higher correctness rates. Moreover, between **52%** to **81%** of generated tests were correct whereas **81%-92%** of the tests had *at least one* passing test case. From these results, we can infer that although all the models could not produce correct tests, they can still be useful in generating at least a few viable input/output pairs.

Table 2: Correct tests percentage for HumanEval and SF110

		GPT-3.5-Turbo	StarCoder	Codex (2K)	Codex (4K)
HE	% Correct	52.3%	81.3%	77.5%	76.7%
	% Somewhat Correct	92.3%	81.3%	87.5%	87.4%
SF110	% Correct	6.9%	51.9%	46.5%	41.1%
	% Somewhat Correct	16.1%	58.6%	62.7%	53.7%

SF110 Results. The correctness rates achieved by the LLMs are rather low. Less than **52%** of the produced tests are correct for all models. Even when considering the unit tests that produced at least one passing test case (*somewhat correct*), only up to **63%** fulfill this criterion. The best-performing model for the SF110 dataset was StarCoder, which produced **51.9%** correct tests. Codex (2K) was the best performing LLM for generating unit tests that have *at least one* passing test case.

4.3 Test Coverage

HumanEval Results. Table 3 shows the line and branch coverage for the HumanEval dataset, computed considering all the Java classes in the dataset. The LLMs achieved line coverage ranging from **67%** to **87.7%** and branch coverage ranging from **69.3%** to **92.8%**. Codex (4K) exhibited the highest line and branch coverage of **87.7%** and **92.8%**, respectively. However, the coverage of the unit tests generated by LLMs are below the coverage reported by the manual tests and those generated by Evosuite. In fact, Evosuite, which relies on an evolutionary algorithm to generate JUnit tests, has a higher line and branch coverage than the manually written tests.

Table 3: Line and branch coverage

	Metric	GPT-3.5-Turbo	StarCoder	Codex-2K	Codex-4K	Evosuite	Manual
HumanEval	Line Coverage	69.1%	67.0%	87.4%	87.7%	96.1%	88.5%
	Branch Coverage	76.5%	69.3%	92.1%	92.8%	94.3%	93.0%
SF110	Line Coverage	1.3%	1.1%	1.9%	1.2%	27.5%	–
	Branch Coverage	1.6%	0.5%	1.1%	0.7%	20.2%	–

SF110 Results. The test coverage for SF110 is worse when compared to HumanEval (they were less than **2%** for all models). Codex (2K) was the best performing one in terms of line coverage (**1.9%**), whereas GPT-3.5-Turbo had the highest branch coverage (**1.6%**). Yet, these coverages are ≈11-19× lower than the coverage achieved by Evosuite's tests.

4.4 Test Smells

HumanEval Results. Table 4 shows that the LLMs produced the following smells[3]: Assertion Roulette (AR) [67], Conditional Logic Test (CLT) [45], Empty Test (EM) [51], Exception Handling (EH) [51], Eager Test (EA) [67], Lazy Test (LT) [67], Duplicate Assert (DA) [51], Unknown Test (UT) [51], , and Magic Number Test (MNT) [45]. We found that Magic Number Test (MNT) and Lazy Test (LT) are the two most reoccurring test smell types across *all* the approaches, *i.e.*, in the unit tests generated by the LLMs and Evosuite as well as the ones created manually. The **MNT** smell occurs when the unit test hard-codes a value in an assertion without a comment explaining it, whereas the **LT** smell arises when multiple test methods invoke the same production code.

Table 4: Test smells distribution for the HumanEval dataset.

Test Smell	Codex (2K)	Codex (4K)	StarCoder	GPT-3.5-Turbo	Evosuite	Manual
AR	61.3%	59.7%	51.3%	23.8%	15.0%	0.0%
CLT	0.0%	0.0%	0.0%	1.5%	0.0%	0.0%
EM	1.9%	1.3%	3.8%	0.8%	0.0%	0.0%
EH	0.0%	0.0%	0.0%	0.0%	100.0%	100.0%
EA	60.6%	59.1%	48.8%	23.8%	16.3%	0.0%
LT	39.4%	41.5%	51.3%	86.2%	99.4%	100.0%
DA	15.6%	14.5%	10.6%	3.1%	0.6%	0.0%
UT	10.0%	5.7%	6.3%	0.8%	0.0%	0.0%
MNT	100.0%	100.0%	100%	100.0%	100.0%	100.0%

Whereas Codex, StarCoder, and GPT-3.5-Turbo did not produce unit tests with the Exception Handling (EH) smell, this smell type was frequent in all manually created tests and those generated by Evosuite. We also found that Assertion Roulette (AR) is a common smell produced by LLMs (frequency between **23.8%** – **61.3%**) and that also occurred in Evosuite in **15%** of its generated tests. This smell occurs when the same test method invokes an `assert` statement to check for different input/output pairs and does not include an error message for each of these asserts. Similarly, the LLMs and Evosuite also produced unit tests with the Eager Test smell (EA), in which a single test method invokes different methods from the production class, as well as the Duplicate Assert smell (DA) (caused by multiple assertions for the same input/output pair).

SF110 Results. The smells detected for the SF110 tests are listed in Table 5. Similar to HumanEval, Magic Number Test (MNT), Assertion Roulette (AR), and Eager Tests (EA) are frequently occurring smells in the unit tests generated by the LLMs and Evosuite. The LLMs generated other types of smells that were not observed for the HumanEval dataset, namely Constructor Initialization (CI) [51], Mistery Guest (MG) [67], Redundant Print (RP) [51], Redundant Assertion (RA) [51], Sensitive Equality (SE) [67], Ignored Test (IT) [51], and Resource Optimism (RO) [51].

While LLMs produced tests that had Empty Tests (EM), Redundant Print (RP), Redundant Assertion (RA), and Constructor Initialization (CI) smells, Evosuite did not generate any unit test with these smell types. We also observed that StarCoder generated (proportionally) more samples than the other models (**96.7%** of its generated tests had at least one test smell).

Table 5: Test smells distribution for the SF110 dataset (RQ1).

Test Smell	GPT-3.5-Turbo	StarCoder	Codex (2K)	Codex (4K)	Evosuite
AR	4.6%	35.1%	14.4%	17.1%	35.0%
CLT	2.3%	2.4%	0.5%	1.3%	0.0%
CI	0.0%	4.9%	0.0%	0.7%	0.1%
EM	0.0%	3.8%	7.2%	1.3%	0.0%
EH	2.3%	18.2%	20.7%	19.1%	91.2%
MG	0.0%	3.5%	2.7%	3.3%	3.0%
RP	0.0%	10.6%	4.5%	5.9%	0.0%
RA	0.0%	0.3%	0.9%	0.7%	0.0%
SE	0.0%	1.9%	0.9%	1.3%	13.7%
EA	12.6%	39.7%	28.4%	31.6%	39.6%
LT	21.8%	33.4%	60.8%	60.5%	46.4%
DA	1.1%	11.7%	1.4%	2.0%	1.5%
UT	0.0%	21.2%	21.2%	10.5%	22.9%
IT	0.0%	0.3%	0.0%	0.0%	0.0%
RO	0.0%	4.6%	2.7%	3.9%	2.7%
MNT	21.8%	95.4%	93.2%	96.1%	91.2%

5 RQ2 RESULTS

Similar to RQ1, we investigated how code elements in a context influence the generated unit tests with respect to their ***compilation status***, ***correctness***, ***coverage***, and ***quality***.

5.1 Compilation Status

Fig. 1 shows the compilation rates for the HumanEval and SF110 datasets across the different scenarios and LLMs.

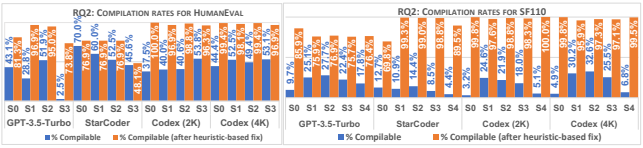

Figure 1: Compilation rates for HumanEval and SF110

[3]We hide *Default Test*, *General Fixture*, *Mystery Guest*, *Verbose Test*, *Resource Optimism*, *Dependent Test*, and other test smell types supported by TsDetect because they did not occur in any of the listed approaches

HumanEval Results. Scenario 3 (**S3**) increased the original (**S0**) compilation rates for Codex (2K and 4K) from **37.5%**, and **44.4%** to **53.8%** and **53.1%**, respectively. Although scenario 3 increased the original compilation rates (blue bars in Fig. 1), these tests have similar heuristic-based fix rates. In the case of StarCoder, the original prompt (**S0**) was the best in generating compilable code. GPT-3.5-Turbo, on the other hand, experienced a sharp decrease from 43.1% to 2.5% for S3. Upon further inspection, we found that scenario 3 triggered GPT-3.5-Turbo 3.5 to include the original class under test in its entirety, followed by the unit test. This resulted in two package declarations on the produced output; one was placed in the very first line (corresponding to the CUT's package), and the other was placed after the CUT for the unit test's package. These duplicated package declarations lead to compilation errors. These issues were later fixed by applying the heuristic **H3**. For the GPT-3.5-Turbo model, the best-performing context scenario was **S2**, in which the prompt does not include sample input/output pairs.

SF110 Results. **S2** increased the original (**S0**) compilation rates for GPT-3.5-Turbo, StarCoder, and Codex (4K). However, scenario 1 (**S1**) was the best performer for Codex (2K), while scenario 2 (**S2**) was the second-best performer. What these results show is that it is beneficial to include a *minimal* context, which contains only the MUT's implementation when generating test cases. The benefit seems twofold: (1) it can increase the compilation rate of the generated code snippets, and (2) it consumes less input tokens, as other methods from the class under test are removed.

5.2 Test Correctness

Fig. 2 shows the percentage of unit tests generated by the LLMs that are *correct* for the HumanEval and SF110 datasets. The best performing scenarios for an LLM are highlighted in green.

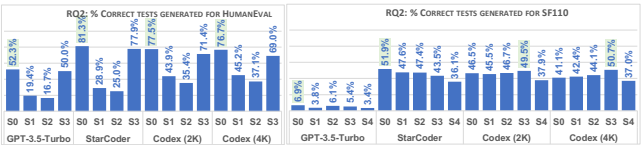

Figure 2: Correctness rates

HumanEval Results. The original context (**S0**) is the one that leads to the highest amount of correct tests for the HumanEval dataset. Among all scenarios, scenario 3 **S3**) had a *similar* correctness rate compared to the original prompt used in RQ1 for GPT-3.5-Turbo and Codex (2K, 4K). It is important to highlight that whereas GPT-3.5-Turbo only had 73.8% compilable tests in scenario 3 (compared to 81.3% tests from the original prompt) it still had a similar correctness rate. Yet, the original prompt is the one that has the highest correctness rates. Recall that scenario 3 (**S3**) is the one in which the implementation of the method under test is not included in the prompt. These results show that LLMs can still generate unit tests even if the implementation is not provided. Such a scenario can be useful in TDD; where developers write tests *before* the production code.

• **Effects on including input/output examples on the prompt.** The HumanEval dataset has input/output examples in its problem description (see Listing 2). Thus, for this dataset, we also investigated to what extent LLMs are able to generate unique input/output pairs that are not included in the original problem description and how these are related to the test correctness rates observed. We manually inspected each generated test to compute the ***total number of unique input/output pairs*** generated. For each unique input/output pair, we compared with the ones provided in the problem's description in order to compute the ***total number of input-output pairs that are from the problem description*** and the ***total number of input-output pairs that are not in the problem description***.

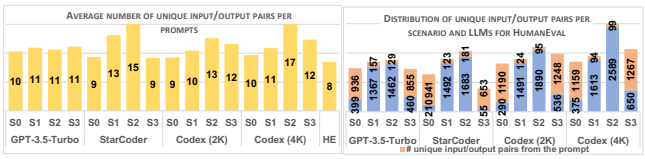

Figure 3: *Left*: Average number of unique input/outputs per prompt for each LLM and the original dataset (HumanEval - HE). *Right*: Total number of unique input/outputs that are and are not from the problem's description.

Fig. 3 (*left*) shows the average number of unique input/output pairs for each LLM and scenario combination compared to the problem description in the HumanEval dataset. Each problem in the HumanEval dataset provides an average of **8** input/output pair examples, whereas the LLMs provide more than that, as the prompts explicitly request 10 test cases for each problem description. We observed that the scenarios **S1** and **S2**, which do not include input-output pairs in the prompt, has a *higher* average of a number of unique input-output pairs.

Fig. 3 (right) shows how many of the generated input/output pairs by the LLMs are from the problem's description and how many are not. We found that the scenarios **S1** and **S2** generated *more* input-output pairs that are not from the original description, whereas the scenarios **S0** and **S3** are repeating the test cases from the prompt. That is, the models are behaving like "parrots" [9] by using the same input/output in the prompt and just formatting it as a test case without generating new examples. When contrasting with the correctness rates observed in Fig. 2 we see that scenarios **S1** and **S2** were consistently lower for all LLMs. These results show that although scenarios **S1** and **S2** generated *more* input-output examples, those were not necessarily correct. The prompts that included examples of input-outputs had *higher* correctness rates.

SF110 Results. While the original prompt (**S0**) achieved the highest correctness rate for GPT-3.5-Turbo (6.9%) and StarCoder (51.9%), the other LLMs observed a correctness increase when using the context from scenario 3 (**S3**). Codex (4K) experienced the highest increase (from 37.9% to 50.7%) for S3. This scenario (**S3**) has a context that only includes the MUT's Javadoc and signature and removes other methods from the class where the MUT is declared.

5.3 Test Coverage

Fig. 4 shows the *line* and *branch* coverage for each scenario.

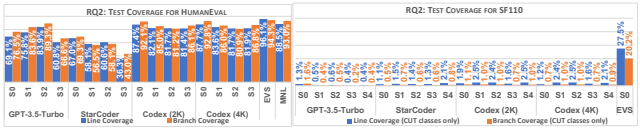

Figure 4: Line and Branch Coverage across different datasets, scenarios, and LLMs (EVS = Evosuite; MNL = Manual).

HumanEval Results. For Codex, scenario 1 is the one that had the highest line coverage among the different scenarios in these models. GPT-3.5-Turbo and StarCoder, on the other hand, had scenario 2 as the one with the highest *line* coverage. With respect to *branch* coverage, we found that scenario 3 was the best performing one for Codex, and scenario 2 is the best one for GPT-3.5-Turbo and StarCoder. None of the scenarios for Codex (2K and 4K) and StarCoder outperformed the line/branch coverage of the original prompts nor the coverage achieved by the manual and Evosuite's tests.

SF110 Results. Among all scenarios, scenario 1 (**S1**) and scenario 2 (**S2**) had a slightly *higher* line coverage when compared to the original prompt (**S0**) used in RQ1 for Codex (2K) and Codex (4K), respectively. For StarCoder the scenario 4 had a higher line coverage than the original one. The original context of GPT-3.5-Turbo, on the other hand, had the highest observed line coverage. In the case of branch coverage, scenario 1 (**S1**) had slightly higher coverage for Codex (4K), whereas scenario 4 (**S4**) was the best one for StarCoder. However, these increases are still much lower than Evosuite's test coverage, which achieved ≈ 27% line and branch coverage.

5.4 Test Smells

HumanEval Results. Table 6 shows the distribution of smells for different scenarios and LLMs. The cells highlighted in green are those in which the percentage is lower than the original context, whereas those highlighted in red have a higher percentage than the original context. In terms of smell types, all scenarios have the same smell types that occurred in the original prompts (see Table 4). We also observe that, overall, the scenarios tended to decrease the incidence of generated smells. When comparing each scenario to one another, there is no clear outperformer across all the LLMs. Yet, Scenario 3 for GPT-3.5-Turbo had higher percentages than the original context, on average. Although the average increases are not significant (0.6% and 0.2% for these LLMs, respectively).

Table 6: Test smells distribution for the HumanEval dataset.

	GPT-3.5-Turbo			StarCoder			Codex (2K)			Codex (4K)		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
AR	7.1%	11.8%	30.5%	36.9%	36.3%	48.1%	16.8%	38.6%	61.0%	16.6%	40.3%	63.2%
CLT	6.5%	3.3%	0.8%	0.0%	0.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
EM	0.0%	0.7%	3.4%	1.9%	8.1%	3.8%	4.5%	3.2%	1.9%	1.3%	1.3%	1.9%
EA	7.1%	10.5%	26.3%	28.8%	30.0%	48.1%	15.5%	37.3%	56.5%	15.3%	38.4%	58.1%
LT	85.2%	92.8%	82.2%	61.9%	63.8%	53.1%	84.5%	60.8%	44.2%	84.7%	60.4%	42.6%
DA	1.3%	0.0%	1.7%	8.1%	11.3%	11.3%	0.6%	8.2%	11.0%	1.9%	6.9%	11.6%
UT	0.0%	0.7%	3.4%	11.3%	13.8%	6.3%	13.5%	16.5%	2.6%	5.1%	8.2%	2.6%
MNT	89.7%	98.7%	100%	99.4%	99.4%	100%	100%	100%	100%	100%	100%	100%

SF110 Results. As shown Table 7, there is not any scenario that consistently outperforms the other. However, we can observe that scenario 2 for GPT-3.5-Turbo produces more test smells than the other scenarios, as we can see from the cells highlighted in red.

6 DISCUSSION

– **LLMs *vs.* Evosuite**: Across all the studied dimensions, LLMs performed *worse* than Evosuite. One reason is that LLMs do not always produce compilable unit tests (Table 1). For example, while Evosuite produced one unit test for each of the 160 classes under test, GPT-3.5-Turbo only produced **130** compilable (*i.e.*, executable) unit tests. Another reason is that LLMs do not seem to pay attention to the current MUT's implementation. A piece of evidence for this is that scenario 3 (which does not include the MUT's implementation) has better compilation rates than the rest. However, we also observed that GPT-3.5-Turbo generated test cases for "stress-testing", *e.g.*, using `Integer.MAX_VALUE` to test for the MUT's behavior in the face of exceptionally large inputs.

Table 7: Test smells distribution for the SF110 dataset (RQ2).

	Codex (2K)				Codex (4K)				StarCoder				GPT-3.5-Turbo			
	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
AR	17.3%	12.8%	12.4%	7.8%	17.5%	13.5%	13.6%	8.3%	23.0%	23.5%	21.4%	27.1%	6.6%	7.8%	4.4%	12.1%
CLT	0.0%	0.5%	0.0%	0.7%	0.0%	0.0%	0.0%	0.8%	1.4%	1.6%	1.4%	1.1%	0.5%	1.7%	1.1%	3.5%
CI	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	0.0%	0.0%	1.1%	0.0%	0.0%	0.0%	0.0%
EM	8.2%	5.1%	24.8%	5.9%	7.7%	5.0%	21.6%	5.4%	1.4%	1.6%	2.9%	2.9%	0.0%	0.0%	1.1%	2.1%
EH	14.3%	19.5%	15.3%	24.5%	15.5%	18.5%	14.1%	25.7%	17.2%	22.5%	25.3%	21.5%	2.2%	3.3%	2.7%	5.0%
MG	2.0%	1.5%	1.0%	2.6%	1.0%	1.5%	1.5%	2.5%	2.2%	2.7%	2.4%	2.7%	1.6%	1.1%	1.1%	3.5%
RP	2.0%	2.1%	4.0%	3.0%	1.5%	2.5%	4.0%	2.9%	6.8%	16.5%	14.1%	10.7%	0.0%	0.0%	0.0%	0.7%
RA	1.0%	0.5%	1.0%	1.5%	0.5%	0.5%	1.0%	1.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.5%	0.7%
SE	1.0%	0.0%	1.5%	1.5%	1.0%	0.5%	1.0%	1.2%	0.6%	0.2%	0.4%	0.9%	0.5%	0.6%	1.1%	2.1%
EA	16.8%	14.4%	11.4%	20.8%	17.0%	13.0%	11.6%	25.3%	24.6%	28.7%	20.8%	35.1%	7.7%	8.3%	6.6%	15.6%
LT	31.6%	44.1%	32.7%	55.8%	33.0%	46.0%	35.2%	57.7%	30.1%	26.0%	27.1%	32.4%	14.2%	16.7%	13.7%	22.0%
DA	6.1%	1.5%	1.5%	1.9%	5.2%	2.5%	2.0%	2.5%	6.4%	4.5%	5.1%	7.2%	2.2%	1.7%	0.5%	2.8%
UT	14.8%	12.3%	30.7%	17.8%	12.9%	10.5%	24.1%	16.6%	17.8%	16.7%	19.4%	20.6%	0.0%	0.0%	1.6%	2.1%
RO	1.5%	1.5%	2.0%	2.2%	1.0%	1.5%	2.5%	2.9%	3.6%	3.3%	3.7%	4.0%	1.6%	1.1%	1.1%	2.8%
MNT	98.5%	98.5%	98.0%	91.8%	97.9%	97.5%	98.5%	95.0%	91.2%	96.9%	99.0%	96.4%	18.6%	21.1%	18.0%	29.1%

– **Codex and StarCoder perform *better* than GPT-3.5-Turbo**. This can be explained by the fact that Codex and StarCoder are LLMs fine-tuned for code-related tasks in contrast to GPT-3.5-Turbo, which is tailored to dialogues (natural language).

– **LLMs often "hallucinate" inexistent types, methods, *etc.*** For both datasets, the most common compilation error was due to missing symbols. For instance, Codex generated inputs whose type were `Tuple`, `Pair`, `Triple`, `Quad`, and `Quint`, which are non-existent in Java's built-in class types.

– **Synergy between LLMs and TDD**. Although LLMs did not achieve coverages or compilation rates comparable to Evosuite, they can still be useful as a starting point for TDD. As we showed in our RQ2, LLMs can generate tests based on the MUT's JavaDoc. However, given the low correctness rates of LLMs, developers would still need to adjust the generated tests manually.

Given these findings, we observe a need for future research to focus on helping LLMs in reason over data types and path feasibility, as well as exploring the combination of SBST and LLMs for TDD. Furthermore, a recent study [71] surveyed 2,000 developers and analyzed anonymous user data, showing that GitHub Copilot makes developers more productive because the generated code can automate repetitive tasks. Thus, our findings provide some initial evidence that *practitioners* following a TDD approach could benefit from LLM-generated tests as a means to speed up their testing. Although further user studies would be needed to verify this hypothesis.

6.1 Threats to Validity

Creating canonical solutions for the Java samples in the HumanEval dataset [5] introduced an internal validity threat. To mitigate it, we extensively vetted our solution with a test set provided by the

dataset. Another validity threat relates to the use of the SF110 benchmark [20], JaCoCo [2] for calculating coverage results and TsDetect [52] to find test smells. In this case, our analyses depend on the representativeness of the SF110 dataset (construct validity threat) and the accuracy of these tools. However, the SF110 dataset is commonly used to benchmark automated test generation tools [12, 20, 59] and JaCoCo and TsDetect are state-of-the-art tools [10, 68].

7 RELATED WORK

Previous works have focused on creating source code that can do a specific task automatically (code generation). The deductive synthesis approach [27, 44], in which the task specification is transformed into constraints, and the program is extracted after demonstrating the satisfaction of the constraints, is one of the foundations of program synthesis [30]. Recurrent networks were used by Yin *et al.* [70] to map text to abstract syntax trees, which were subsequently coded using attention. A variety of large language learning models have been made public to generate code (e.g., CodeBert [19], CodeGen [47] and CodeT5 [69]) after being refined on enormous code datasets. Later, GitHub Copilot developed an improved autocomplete mechanism using the upgraded version of Codex [16], which can help to solve fundamental algorithmic problems [18]. Our work focuses not on code generation but on how a publicly available code generation tool can be used for specialized tasks like unit test generation without fine-tuning (*i.e.*, zero-shot test generation).

Shamshiri *et al.* [60] proposed a search-based approach that automatically generates tests that can reveal functionality changes, given two program versions. On the other hand, Tufano *et al.* [66] proposed an approach that aims to generate unit test cases by learning from real-world focal methods and developer-written test cases. Pacheco *et al.* [48] presented a technique that improves random test generation by incorporating feedback obtained from executing test inputs as they are created for generating unit tests. Lu *et al.* [43] worked on testing autonomous driving systems with reinforcement learning. Lima et [41] surveyed the practitioners on software testing and refactoring. In our work, we focus on zero-shot unit test generation using different contexts in order to measure the LLM's ability to generate compilable, correct, and smell-free tests.

Schäfer *et al.* [57] used Codex [16] to automatically generate unit tests using an adaptive approach. They used 25 npm packages to evaluate their tool, TESTPILOT. However, they evaluated their model only on statement coverage. They did not provide insight into the quality of the generated test cases and the choice of using a specific prompt structure. Lemieux *et al.* [38] combined the Search-based software testing (SBST) technique with the LLM approach. It explored whether Codex can be used to help SBST's exploration. Nashid *et al.* [46] aimed to devise an effective prompt to help large language models with different code-related tasks, *i.e.*, program repair and test assertion generation. Their approach provided examples of the same task and asked the LLM to generate code for similar tasks. Li *et al.* [40] used ChatGPT [1] to find failure-inducing tests with differential prompting. Bareiß *et al.* [7] performed a systematic study to evaluate how a pre-trained language model of code, Codex, works with code mutation, test oracle generation from natural language documentation, and test case generation using few-shot prompting like Nashid *et al.* [46]. However, the benchmark has only 32 classes, so the findings may not be generalized. This work provides direction toward using examples of usage or similar tasks as a context. However, in a real case, there may not be any example of using the method and class that can be used in the prompt, and creating an example of a similar task needs human involvement. Our work focused on different contexts taken from the code base. We evaluated the quality of the generated unit tests not only on coverage and correctness but also based on the presence of test smells.

8 CONCLUSION

We studied the capability of three code generation LLMs for unit test generation. We conducted experiments with different contexts in the prompt and compared the results based on compilation rate, test correctness, coverage, and test smells. These models have a close performance with the state-of-the-art test generation tool for the HumanEval dataset, but their performance is poor for open-source projects from Evosuite based on coverage. Though our developed heuristics can improve the compilation rate, several generated tests were not compilable. Moreover, they heavily suffer from test smells like Assertion Roulette and Magic Number Test. In future work, we will explore how to enhance LLMs to understand language semantics better in order to increase test correctness and compilation rates.

REFERENCES

[1] 2023. Chat completions. Accessed Mar 25, 2023. https://platform.openai.com/docs/guides/chat

[2] 2023. JaCoCo - Java Code Coverage Library. https://www.jacoco.org/jacoco/trunk/index.html [Online; accessed 30. Mar. 2023].

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[4] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, ..., and Bing Xiang. 2022. Multi-lingual Evaluation of Code Generation Models. (2022).

[5] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, et al. 2022. Multi-lingual Evaluation of Code Generation Models.

[6] A. Bacchelli, P. Ciancarini, and D. Rossi. 2008. On the effectiveness of manual and automatic unit test generation. In *2008 The Third Int'l Conf. on Software Engineering Advances*. IEEE, 252–257.

[7] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel. 2022. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code. *arXiv preprint arXiv:2206.01335* (2022).

[8] K. Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.

[9] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big?. In *Proc'd. of the 2021 ACM Conf. on fairness, accountability, and transparency*. 610–623.

[10] I. Bilal, I. Al-Taharwa, S. Rami, I. M. Alkhawaldeh, and N. Ghatasheh. 2021. JaCoCo-Coverage Based Statistical Approach for Ranking and Selecting Key Classes in Object-Oriented Software. *J. Eng. Sci. Technol* 16 (2021), 3358–3386.

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901.

[12] D. Bruce, H. D. Menéndez, and D. Clark. 2019. Dorylus: An ant colony based tool for automated test case generation. In *Search-Based Software Engineering: 11th Int'l Symposium, SSBSE 2019, Tallinn, Estonia, August 31–September 1, 2019, Proc'd. 11*. Springer, 171–180.

[13] J. Campos, A. Arcuri, G. Fraser, and R. Abreu. 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proc'd. of the 29th ACM/IEEE inter'l Conf. on Automated software engineering*. 55–66.

[14] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* (2023).

[15] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[16] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[17] Ermira Daka, José Campos, G. Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015). https://doi.org/10.1145/2786805.2786838

[18] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming, et al. 2022. GitHub Copilot AI pair programmer: Asset or Liability? *arXiv preprint arXiv:2206.15331* (2022).

[19] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547.

[20] G. Fraser and A. Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proc'd. of the 19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419.

[21] G. Fraser and A. Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *34th Int'l Conf. on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 178–188.

[22] G. Fraser and A. Arcuri. 2014. A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[23] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 1–49.

[24] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. t. Yih, L. Zettlemoyer, and M. Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022).

[25] Y. Gao and C. Lyu. 2022. M2TS: Multi-Scale Multi-Modal Approach Based on Transformer for Source Code Summarization. *arXiv preprint arXiv:2203.09707* (2022).

[26] D. Gonzalez, J. C. S. Santos, A. Popovich, M. Mirakhorli, and M. Nagappan. 2017. A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension. In *2017 IEEE/ACM 14th Int'l Conf. on Mining Software Repositories (MSR)*. IEEE, 391–401.

[27] C. Green. 1969. Application of Theorem Proving to Problem Solving. In *Proc'd. of the 1st Int'l Joint Conf. on Artificial Intelligence* (Washington, DC) *(IJCAI'69)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 219–239.

[28] M. Greiler, A. Zaidman, A. van Deursen, and M.-A. Storey. 2013. Strategies for Avoiding Text Fixture Smells during Software Evolution. In *Proc'd. of the 10th Working Conf. on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, 387–396.

[29] E. M. Guerra and C. T. Fernandes. 2007. Refactoring Test Code Safely. In *Int'l Conf. on Software Engineering Advances (ICSEA 2007)*. 44–44.

[30] S. Gulwani, O. Polozov, R. Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[31] M. A. Hadi, I. N. B. Yusuf, F. Thung, K. G. Luong, J. Lingxiao, F. H. Fard, and D. Lo. 2022. On the Effectiveness of Pretrained Models for API Learning. In *Proc'd. of the 30th IEEE/ACM Int'l Conf. on Program Comprehension* (Virtual Event) *(ICPC '22)*. ACM, New York, NY, USA, 309–320.

[32] M. Hilton, J. Bell, and D. Marinov. 2018. A Large-Scale Study of Test Coverage Evolution. In *Proc'd. of the 33rd ACM/IEEE Int'l Conf. on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 53–63.

[33] M. Ivanković, G. Petrović, R. Just, and G. Fraser. 2019. Code Coverage at Google. In *Proc'd. of the 2019 27th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 955–963.

[34] M. Izadi, R. Gismondi, and G. Gousios. 2022. CodeFill: Multi-token Code Completion by Jointly Learning from Structure and Naming Sequences. In *44th Int'l Conference on Software Engineering (ICSE)*.

[35] S. Kim, J. Zhao, Y. Tian, and S. Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd Int'l Conf. on Software Engineering (ICSE)*. IEEE, 150–162.

[36] P S Kochhar, F Thung, N Nagappan, T Zimmermann, and D Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[37] T. Koomen and M. Pol. 1999. *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing*. Addison-Wesley Longman Publishing Co., Inc., USA.

[38] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th Int'l Conf. on Software Engineering, ser. ICSE*.

[39] R. Li, L. Ben allal, Y. Zi, N. Muennighoff, D. Kocetkov, ..., and H. de Vries. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research* (2023). Reproducibility Certification.

[40] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *2023 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 14–26.

[41] D. L. Lima, R. De Souza Santos, G. P. Garcia, S. S. Da Silva, C. França, and L. F. Capretz. 2023. Software Testing and Code Refactoring: A Survey with Practitioners. In *2023 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME)*. 500–507.

[42] S. P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Inf. Theory* 28 (1982), 129–136.

[43] C Lu, T Yue, M Zhang, and S Ali. 2023. DeepQTest: Testing Autonomous Driving Systems with Reinforcement Learning and Real-world Weather Data. *ACM Transactions on Software Engineering and Methodology* (2023).

[44] Z. Manna and R. J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (mar 1971), 151–165.

[45] G. Meszaros, S. M. Smith, and J. Andrea. 2003. The Test Automation Manifesto. In *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, F. Maurer and D. Wells (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–81.

[46] N. Nashid, M. Sintaha, and A. Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. *ICSE23* (2023).

[47] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv preprint* (2022).

[48] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. 2007. Feedback-directed random test generation. In *29th Int'l Conf. on Software Engineering (ICSE'07)*. IEEE, 75–84.

[49] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia. 2016. On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study. In *2016 IEEE/ACM 9th Int'l Workshop on Search-Based Software Testing (SBST)*. 5–14.

[50] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 980–994.

[51] A. Peruma, K. Almalki, C. D. Newman, M. Wiem Mkaouer, A. Ouni, and F. Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proc'd. of the 29th Annual Int'l Conf. on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 193–202.

[52] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. 2020. TsDetect: An Open Source Test Smells Detection Tool. In *Proc'd. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1650–1654.

[53] J. A. Prenner, H. Babii, and R. Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *2022 IEEE/ACM Int'l Workshop on Automated Program Repair (APR)*. 69–75.

[54] P. J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65.

[55] P. Runeson. 2006. A survey of unit testing practices. *IEEE software* 23, 4 (2006), 22–29.

[56] J. Savelka, A. Agarwal, C. Bogart, Y. Song, and M. Sakr. 2023. Can Generative Pre-Trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?. In *Proc'd. of the 2023 Conf. on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. ACM, New York, NY, USA, 117–123.

[57] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. 2023. Adaptive Test Generation Using a Large Language Model. *arXiv preprint arXiv:2302.06527* (2023).

[58] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli. 2019. On the effectiveness of manual and automatic unit test generation: ten years later. In *2019 IEEE/ACM 16th Int'l Conf. on Mining Software Repositories (MSR)*. IEEE, 121–125.

[59] M. M. D. Shahabi, S. P. Badiei, S. E. Beheshtian, R. Akbari, and S. M. R. Moosavi. 2017. On the performance of EvoPSO: A PSO based algorithm for test data generation in EvoSuite. In *2017 2nd Conf. on Swarm Intelligence and Evolutionary Computation (CSIEC)*. IEEE, 129–134.

[60] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser. 2018. How Do Automatically Generated Unit Tests Influence Software Maintenance?. In *2018 IEEE 11th Int'l Conf. on Software Testing, Verification and Validation (ICST)*. 250–261.

[61] Inbal Shani. 2023. Survey reveals AI's impact on the developer experience | The GitHub Blog. *GitHub Blog* (June 2023). https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/#methodology
[62] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*. 71–82.
[63] M. L. Siddiq, A. Samee, S. R. Azgor, M. A. Haider, S. I. Sawraz, and J. C. S. Santos. 2023. Zero-shot Prompting for Code Complexity Prediction Using GitHub Copilot. In *2023 The 2nd Intl. Workshop on NL-based Software Engineering*.
[64] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th Int'l Conf. on Mining Software Repositories (MSR)*. IEEE, 329–340.
[65] Dave A. Thomas and A. Hunt. 2002. Mock Objects. *IEEE Softw.* 19 (2002), 22–24. https://doi.org/10.1109/MS.2002.1003449
[66] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
[67] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. 2001. Refactoring Test Code. In *Proc'd. 2nd Int'l Conf. on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, M. Marchesi and G. Succi (Eds.).
[68] T. Virgínio, L. Martins, R. Santana, A. Cruz, L. Rocha, H. Costa, and I. Machado. 2021. On the test smells detection: an empirical study on the JNose test accuracy. *Journal of Software Engineering Research and Development* 9 (2021), 8–1.
[69] Y. Wang, W. Wang, S. Joty, and S. C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proc'd. of the 2021 Conf. on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708.
[70] P. Yin and G. Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proc'd. of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 440–450.
[71] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian. 2022. Productivity Assessment of Neural Code Completion. In *Proc'd. of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. ACM, New York, NY, USA, 21–29.