Exploring the Effectiveness of Large Language Models in Generating Unit Tests

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

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

Ridwanul Hasan Tanvir
Bangladesh University of Eng. and Tech.
Dhaka, Bangladesh
1705016@ugrad.cse.buet.ac.bd

Noshin Ulfat
Bangladesh University of Eng. and Tech.
Dhaka, Bangladesh
1705089@ugrad.cse.buet.ac.bd

Fahmid Al Rifat
Bangladesh University of Eng. and Tech.
Dhaka, Bangladesh
1705087@ugrad.cse.buet.ac.bd

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

***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. To fill this gap, we investigated how well three generative models (CodeGen, Codex, and GPT-3.5) can generate test cases. We used two benchmarks (HumanEval and Evosuite SF110) to investigate the context generation's effect in the unit test generation process. We evaluated the models based on compilation rates, test correctness, 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 SF110 benchmark. The generated tests also suffered from test smells, such as Duplicated Asserts and Empty Tests.**

***Index Terms*—code generation, unit testing, large language models, test smells, test generation**

I. Introduction

Automated code generation approaches generate code from *prompts* [1]. These prompts specify the developer's intent and have varying granularity and structure. They may include sentences, code comments, code elements (*e.g.*, function signatures, expressions, *etc.*), or a combination of these. Hence, developers can write an initial code and/or comment and rely on these tools to generate the remaining code to speed up the software development process [2]. With the release of GitHub Copilot[1] in 2021, these techniques are increasingly being adopted in the industry [2]. GitHub Copilot relies on a transformer-based [3] Large Language Model (LLM) fine-tuned for code generation.

With the increasing popularity of LLMs prior works have investigated the correctness of the generated code [4], their quality (in terms of code smells) [5], security [6] as well as whether it can be used for API learning tasks [7], and code complexity prediction [8]. However, it is currently unclear the effectiveness of using prompt-based pre-trained code generation models for generating *unit tests*.

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

Unit testing is an important software maintenance activity because it helps developers identify and fix defects early on in the development process before they can propagate and cause more significant problems [9]–[11]. Moreover, unit tests help developers understand how the various code units in a software system fit together and work as a cohesive whole [12]. Given its importance, prior works (*e.g.*, [13]) developed automated test case generation techniques.

To better understand the current capabilities of LLMs in generating unit tests, we conducted an empirical study using three LLMs (Codex [14], ChatGPT3.5 [15] and CodeGen [16]) to generate JUnit5 tests for classes in the HumanEval dataset's Java version [17] and in 47 open-source projects from the SF110 dataset [13]. We answered two research questions. In the first question, we used the full class under test as a context for the LLMs to generate unit test cases. In the second research question, we examine how different context styles (*e.g.*, only using the method under test, the presence, and absence of JavaDoc *etc.*) can influence the generated tests. We examined the produced tests with respect to their compilation rates, correctness, code coverage, and test smells.

The **contributions** of our work are:

- A systematic study of three LLMs for zero-shot unit test generation for 194 classes from 47 open-source projects in the SF110 dataset [18] and 160 classes from the HumanEval dataset [17].
- An investigation of the quality of the produced unit tests by studying the prevalence of test smells in the generated unit tests by different code generation models.
- A comparison of how different context styles affect the performance of LLMs in generating tests.
- A thorough discussion about the implication of using code generation models for unit test generation in a Test Driven Development (TDD) environment.
- A replication package with all the scripts used to gather the

data and spreadsheets compiling all the results[2].

II. Background

In this section, we define the core concepts needed for our paper to be understood.

A. Unit Tests & Test Smells

The goal of *unit testing* is to validate that each program unit is working as intended and meets the specified requirements [19]. A *unit* refers to a piece of code that can be separated and examined independently (*e.g.*, functions, methods, classes, *etc.*). In this paper, *classes* are our units under test.

Just like production code, unit tests need to be not only *correct* but also satisfy other quality attributes, such as *maintainability* and *readability* [20]. *Unit test smells* (henceforth "test smells") are indicators of potential problems, inefficiencies, or bad programming/design practices in a unit test suite [21]–[25]. They are often subtle and may not necessarily result in immediate failures or defects, but they can significantly impact the maintainability and effectiveness of the test suite over time [26]. 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.

For example, the Java code in Listing 1 has a unit test for a method (`largestDivisor(int)`). 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 [26]. It also contains multiple test cases under the same test methods, an example of *Assertion Roulette* smell [24].

LargestDivisorTest.java

```java
@Test
void testLargestDivisor() {
    assertEquals(5, LargestDivisor.largestDivisor(15));
    assertEquals(1, LargestDivisor.largestDivisor(3));
    assertEquals(1, LargestDivisor.largestDivisor(7));
}
```

Listing 1: Example of Unit Test and Unit Test Smell

B. Code Generation

Code generation techniques automatically generate source code from a given *prompt* [1], 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.*The task of generating source code is also known as *sequence-to-sequence (seq2seq) learning* problem [27]. Researchers and practitioners tried to solve the seq2seq problem using Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) based neural network [27], [28]. Attention-based transformer architectures have recently been popular for solving this problem [3]. The transformer is a deep learning model built on an encoder-decoder architecture that leverages the self-attention mechanism to weigh the importance of each input data point [3]. Several well-known language models, like BERT (Bidirectional Encoder Representations from Transformers) [29], T5 (Text-to-Text Transformer) [30] and GPT-3 (Generative Pre-trained Transformer) [31], are powered by the transformer architecture.

These large language models (LLMs) can be trained with source code to help with code-related tasks, *e.g.*, code completion [32]–[34], search [35], and summarization [36]. Examples of code-related LLMs include CodeBERT [35], CodeT5 [37], and CodeGen [16]. They take tokenized textual input and generate an output based on it. However, they have context size limitations (*i.e.*, they accept/generate up to a certain amount of tokens). For example, CodeGen [16] model can process up to 2,048 tokens.

III. Methodology

We address two research questions to understand the effectiveness of LLMs in generating unit tests.

RQ1 *How well can LLMs generate unit tests?*

We used ChatGPT 3.5, Codex, and CodeGen to generate unit tests for competitive programming assignments from the extended version of the HumanEval dataset created by the AWS AI Labs [17] as well as 47 open-source projects from the EvoSuite SF110 benchmark dataset [13]. We measured the LLMs' performance by computing branch/line coverage, and the occurrence of test smells. We also compared the performance of these models with Evosuite [13], an existing state-of-the-art approach.

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

When developers use LLMs to generate unit tests, they create a ***prompt*** (*e.g.*, *"Write a unit test to verify that* `login(req)` *works correctly."*) and the 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 in this question. To answer this question, we conducted a controlled experiment in which we created 3 different scenarios for the HumanEval [14], [17], and 4 scenarios for 47 open-source projects from the EvoSuite SF110 dataset [13]. Each scenario contains a different set of code elements. Then, we use CodeGen, ChatGPT-3.5, and Codex 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.

A. RQ1: LLM-based Unit Test Generation

To investigate whether existing models are capable of generating unit tests, we followed a three-step systematic process: ① we collected **160** Java classes from the **multilingual version of the HumanEval dataset** [17] and **194** Java classes from **47** projects in the **Evosuite SF110 benchmark dataset** [18],

[2]https://doi.org/10.5281/zenodo.7875623

[38]; 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 [13].

1) Data Collection: We retrieved Java classes from:

– The **multilingual HumanEval dataset** [17] contains **160** *prompts* for Java and other programming languages crafted from the original Python-based HumanEval dataset [14]. However, this multilingual version does not provide an implementation for each prompt (*i.e.*, a *canonical solution*). Hence, we wrote the solution for each problem and tested our implementation using the provided test cases. Our solution is encapsulated in a class as a `public static` method. Listing 2 shows a sample taken from this dataset[3], where the prompt is in lines 1–12 and the solution is in lines 13–16.

```java
                              TruncateNumber.java
 1 import java.io.*;
 2 import java.util.*;
 3 import java.math.*;
 4 class TruncateNumber {
 5   /**
 6    * Given a positive floating point number, it can be decomposed into an integer
 7    * part (largest integer smaller than the given number) and decimals (leftover
 8    * part always smaller than 1). Return the decimal part of the number.
 9    * >>> truncateNumber(3.5)
10    * 0.5
11    */
12   public static Double truncateNumber(Double number) {
13     return Math.round( (number - Math.floor(number))
14                         * 1000.0) / 1000.0;
15   }
16 }
```

Listing 2: Sample from the extended HumanEval [17]

– The **SF110 dataset**, which is an Evosuite benchmark consisting of 111 open-source Java projects (though it is named SF110)[4] retrieved from SourceForge. This benchmark contains 23,886 classes, over 800,000 bytecode-level branches, and 6.6 million lines of code [39].

We use the multilingual HumanEval [17] because it has been widely used in prior works to evaluate large language models trained on code [5], [16], [40], [41]. Similarly, we use the SF110 dataset because it is a popular benchmark for unit test generation techniques. Since its publication in 2014, [39], the paper has been cited around 200 times in the last week of March 2023 alone, according to Google Scholar.

Class and Method Under Test Selection: All the classes in the multilingual HumanEval are selected as Classes Under Test (CUTs). Each class has one `public static` method. They may also contain private "helper" methods to aid the solution implementation. Hence, all the public static methods are our MUTs.

For the Evosuite benchmark, we first retrieved only the classes that are `public` and not `abstract`. We then discarded classes placed on a src/test folder, or that contains the keyword "Test" in its name (*i.e.*, test classes). Subsequently, we identify 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 in the end.

2) Unit Test Generation: We used three LLMs to generate unit tests. The first LLM, **Codex**, is a 12 billion parameters LLM [14] descendant of the GPT-3 model [31]. It powers the GitHub Copilot, which is available for writing source code in different programming languages. In this study, we used "code-davinci-002" (the most powerful codex model version of Codex).

The second model, **CodeGen**, is a group of models for synthesizing programs using autoregressive languages [16]. It has three versions for code generation: (1) CodeGen-NL: mainly trained in natural language, (2) CodeGen-Mono: trained in Python code; and (3) CodeGen-Multi: trained on C, C++, Go, Java, JavaScript, and Python. We used the CodeGen-Multi model with 350 million parameters. The third LLM, **GPT-3.5-turbo**, powers the ChatGPT chatbot. It allows multi-turn conversation (*i.e.*, back-and-forth interactions) and can be instructed to generate code. Hence, in this paper, we will use ChatGPT3.5 and GPT3.5 interchangeably. The ChatGPT-3.5 API has three roles: *system*, *assistant*, and *user*. It takes instruction from the *user* role, and the *assistant* gives an answer. The *system* role is meant to help set the behavior of the *assistant* (*e.g.*, "You are a helpful assistant"). However, its API documentation indicates that the model does not pay strong attention to messages from the *system* role [15].

Thus, to generate tests using these LLMs, 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

[3]The code formatting is modified for a better presentation.

[4]Upon further inspection, we noticed two projects have the same ID (82).

from the JUnit5 API. Listing 3 illustrates the structure of these prompts and context, in which lines 1-11 (highlighted in blue) and lines 12–24 are part of the *context* and *prompt*, respectively. The context starts with a comment indicating the CUT's file name followed by the CUT's 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 CUT 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 CUT. It also has import statements to the `@Test` annotation and the `assert*` methods (*e.g.*, `assertTrue(...)`) from JUnit5. Subsequently, the prompt contains 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}{
7          /* ... method implementation ... */
8      }
9      /* ... code after the method under test  ... */
10 }
11 
12 // ${className}${suffix}Test.java
13 ${packageDeclaration}
14 ${importedPackages}
15 import org.junit.jupiter.api.Test;
16 import static org.junit.jupiter.api.Assertions.*;
17 
18 /**
19 * Test class of {@link ${className}}.
20 * It contains ${numberTests} unit test cases for the
21 * {@link ${className}#${methodSignature}} method.
22 */
23 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 leverage the OpenAI API to use the **Codex** model. This LLM 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. 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 as input and generate up to 2,048 tokens. Hence, 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, since it has three roles 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 is the one that provides the generated test code.
- **CodeGen-Multi** can be used with the HuggingFace library[5] (`codegen-350M-multi`). This LLM can only work with a total of 2,048 tokens, which includes the prompt, context, and generated code. Hence, compared to the Codex and GPT-3.5-Turbo, CodeGen has a more limited input/output size. For this reason, we asked the CodeGen model to generate *one* test case for each method under test, but we generated *ten* suggestions by enabling sampling of the model. This way, we have ten suggestions for each prompt. We also keep the same inference parameters as the Codex model, except the temperature, which is not allowed by the HuggingFace API to be set to zero. Instead, we set it to `0.00001` to keep it as close to zero as possible.

3) Data Analysis: 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: `"\n\n// {CUT_classname}"`, `"\n```\n\n##"`, `"</code>"`.

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 pass, or it finds the class declaration (*i.e.*, "`class ABC`"), whichever condition occurs first.

Subsequently, we ran these tests using JUnit5 with JaCoCo [42] to compute coverage and correctness metrics. **Line Coverage** measures how many lines were executed by the

[5]https://huggingface.co

unit test out of the total number of lines [43], [44], *i.e.*, $\frac{Number\ of\ executed\ lines}{Total\ number\ of\ lines} \times 100$.

Branch Coverage is the most well-known and practiced metric in software testing [43] and measures how many branches are covered by a test. It is computed as: $\frac{Number\ of\ visited\ branches}{Total\ number\ of\ branches} \times 100$.

Test Correctness measures how effectively an LLM generates correct input/output pairs. This study assumes that the code under test is implemented correctly. The reasoning behind this assumption is twofold: the HumanEval dataset contains common problems with well-known solutions (which we wrote and tested ourselves), 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 [25], [45]. Due to space constraints, we provide a list of the test smells detectable by TsDetect with their description and source in our replication package.

B. RQ2: Code Elements in a Context

To investigate how different code elements in a context influence the generated unit test, we first created *scenarios* for each of the 160 CUTs collected in RQ1 from the multilingual HumanEval dataset [17] and the 194 CUTs from the 47 Java projects from the EvoSuite benchmark dataset [18], [38]. Next, we generated JUnit5 tests for each scenario. Lastly, we computed the same metrics as in RQ1 (Section III-A3).

1) Scenario Creation: We 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, the three scenarios (**S1–S3**) are created as follows:

S1 It does not contain any JavaDoc (*e.g.*, the JavaDoc from lines 5-11 within Listing 2 is removed from the CUT).
S2 The JavaDoc does not include input/output examples, only the method's behavior description (*e.g.*, Listing 2 will not have lines 9 and 10).
S3 The MUT does not include its implementation, only its signature (*e.g.*, Listing 2 will not have lines 13 and 14).

The first two scenarios demonstrate the effect of changing the JavaDoc elements. Test-Driven Development (TDD) [46] inspires the last scenario approach, where test cases are written before the code implementation.

SF110 Scenarios: Unlike HumanEval, the CUTs from SF110 do not necessarily include input/output pairs. Thus, we generated scenarios slightly different than before:

S1 It removes (i) any code within the CUT *before* and *after* the method under test as well as (ii) the MUT's JavaDoc.
S2 It is the same as S1, but *including* the JavaDoc for the method under test.
S3 It is the same as S2, except that there is no method implementation for the MUT (only its signature).
S4 It mimics S3, but it also includes all the fields and the signatures for the other methods/constructors in the CUT.

S1 and S2 demonstrate the effect of having or not having code documentation (JavaDoc). S3 aims to verify the usefulness of LLMs for TDD whereas S4 is used to understand how other CUT elements are helpful for test generation.

We followed the same models and steps outlined in Section III-A to generate the unit tests. That is, we generated unit tests for each MUT and scenario combination. Then, we used JUnit5, JaCoCo, and TsDetect to measure test coverage, correctness, and quality. Similar to RQ1, we also compared the results to Evosuite [13].

IV. RQ1 Results: Unit Test Generation using LLMs

We analyze the generated unit tests according to four dimensions: **(i)** ***compilation status***; **(ii)** *correctness*; **(iii)** ***coverage***; and **(iv)** ***quality*** (in terms of test smells).

A. Compilation Status

As shown in the second column of Table I, less than 50% of the generated unit tests for the classes in HumanEval are compilable across all the studied models. Only 23.8% of the unit tests generated by CodeGen are compilable. In contrast, Codex and ChatGPT-3.5 had higher compilation rates ($\approx$ 1.5-1.8 times higher than CodeGen). Codex (4K) was the model with the highest compilation rate (44.4%). For the SF110 dataset, the compilation rates are even lower. Between 2.7% and 21% of the generated unit tests for the SF110 dataset are compilable across all the studied LLMs. CodeGen was the LLM that generated the highest amount of compilable tests (21%), whereas Codex had the lowest compilation rate (2.7% and 3.4% for 2K and 4K versions, respectively).

After applying the heuristics described in Section III-A2, we observed that we were able to automatically fix several unit tests (as shown in Table I). For HumanEval tests, the heuristic-based fixes increased the compilation rates by over 55% for Codex, by 38% for ChatGPT, and by 6% for CodeGen for HumanEval. The compilation rates of Codex (2K and 4K) and ChatGPT-3.5 increased from 37.5%, 44.4%, 43.1% to 100%, 99.4%, 81.3%, respectively. In the case of CodeGen, however, not as many samples were repaired; the increase was from 23.8% to 33.1%. This is because CodeGen's unit tests had far more complex compilation errors, such as assigning arrays to collections (and vice-versa), missing identifiers (*i.e.*, the compiler cannot find the symbol) *etc.* For the SF110-related tests,

TABLE I
COMPILATION STATUS OF THE GENERATED UNIT TESTS

	LLM	% Compilable	% Compilable (after fix)	# Test Methods	# Test Files
HumanEval	**ChatGPT 3.5**	43.1%	81.3%	1,117	130
	CodeGen	23.8%	33.1%	844	529
	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	**ChatGPT 3.5**	9.7%	85.9%	194	87
	CodeGen	21.0%	58.5%	83	139
	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

the compilation rates increased by over 70% for ChatGPT-3.5 and Codex and by 37.5% for CodeGen. Once again, CodeGen was the model with more complex compilation errors that could not be removed through heuristics. Whereas over 74% and 85% of the unit tests for ChatGPT3.5 and Codex are compilable, CodeGen only has a total of 58.5% compilable tests. The number of unit tests and test methods is shown in the last two columns of Table I. It can be observed that CodeGen has more test files than test methods. It is because some test files contain no test method but a functional test class.

In fact, the unit tests that were not fixable through heuristics were those that contained *semantic* errors that failed the compilation. We collected all the compilation errors and clustered them using K-means. We used the silhouette method to find the number of clusters K ($K = 48$). Upon inspecting these 48 clusters and making manual adjustments to clusters to correct imprecise clustering, we observed that the top 3 compilation errors for HumanEval were caused by unknown symbols (*i.e.*, the compiler cannot find the symbol), incompatible types, and no suitable method/constructor found for an invocation/instantiation. Among the errors caused by incorrect method invocations, 51% of them were invocations to an assertion method, *e.g.*, `assertEquals()`. In the case of SF110 tests, the top 3 compilation errors were unknown symbols (*i.e.*, the compiler cannot find the symbol), class is not abstract and does not override abstract method , and class is abstract; cannot be instantiated. This differs from what we observed in HumanEval; two reoccurring problems are related to inheritance/polymorphism.

Consequently, for HumanEval, we obtained a total **3,432** test methods (*i.e.*, a method with an `@Test` annotation) scattered across **978** compilable Java test files. For SF110, we had **2,022** test methods and **600** compilable tests. The breakdown per model and dataset is shown in Table I. For comparison, we run Evosuite [13] (with default configuration parameters) to generate unit tests for each of the CUTs. Moreover, in the case of HumanEval, we manually created a JUnit5 test for each input/output pair provided in each prompt (one test method per input/output pair). It is worth highlighting that whereas Codex and ChatGPT generated *one* unit test suggestion per prompt, CodeGen generated *ten* suggestions per prompt. Thus, CodeGen is expected to produce more test files than the number of prompts.

B. Test Correctness

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. Both metrics are reported in Table II.

1) Results for HumanEval dataset: Codex (2K) generated the highest amount of correct unit tests (≈78%), whereas CodeGen generated the least amount of correct unit tests (24%). It is worth mentioning that although ChatGPT 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%). From these results, we can infer that although all the models could not produce correct tests, they could still be useful in generating at least a few viable input/output pairs. We also found that increasing Codex's token size did not yield higher correctness rates.

TABLE II
PERCENTAGE OF CORRECT TESTS FOR HUMANEVAL (HE) AND SF110

		ChatGPT-3.5	CodeGen	Codex (2K)	Codex (4K)
HE	**% Correct**	52.3%	23.9%	77.5%	76.7%
	% Somewhat Correct	92.3%	32.8%	87.5%	87.4%
SF110	**% Correct**	6.9%	30.2%	46.5%	41.1%
	% Somewhat Correct	16.1%	30.2%	62.7%	53.7%

2) Results for SF110 dataset: The best-performing model for the SF110 dataset was Codex (2K) which produced 46.5% correct tests. Yet, the achieved correctness rates are rather low. Less than 50% of the produced tests are correct. CodeGen was the least performing model, producing only 6.9% correct tests. Even when considering the unit tests that produced at least one passing test case (somewhat correct), only up to 63% fulfill this criterion. Once again, Codex (2K) was the best-performing LLM, whereas CodeGen was the worst.

C. Test Coverage

We measured the generated unit tests' line and branch coverage and compared them with the coverage for the tests generated by Evosuite [13]. For HumanEval, we also compared the coverage of the manually created tests.

1) Results for HumanEval dataset: Table III shows the line and branch coverage for the HumanEval dataset, computed considering all the CUTs in the dataset. The results show that the LLMs achieved line coverage ranging from **58.2%** to **87.7%** and branch coverage ranging from **54.7%** 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 III
LINE AND BRANCH COVERAGE

	Metric	ChatGPT-3.5	CodeGen	Codex (2K)	Codex (4K)	Evosuite	Manual
HumanEval	Line Coverage	69.1%	58.2%	87.4%	87.7%	96.1%	88.5%
	Branch Coverage	76.5%	54.5%	92.1%	92.8%	94.3%	93.0%
SF110	Line Coverage	0.1%	0.5 %	2.5 %	1.8%	27.5%	–
	Branch Coverage	0.0%	0.0 %	1.4%	1.4%	27.8%	–

2) Results for SF110 dataset: The test coverage for SF110 is drastically worse when compared to HumanEval. In fact, ChatGPT and CodeGen both achieved a 0% branch coverage and less than 1% line coverage. Among the LLMs, Codex (2K) was the best performing one with 2.5% and 1.4% line and branch coverage. Yet, these coverages are ≈11-19× lower than the coverage achieved by Evosuite's tests.

D. Test Smells

1) Results for HumanEval dataset: Table IV shows the distribution of test smells in different LLMs[6]. The LLMs produced the following smells: Assertion Roulette (AR) [24], Conditional Logic Test (CLT) [26], Constructor Initialization (CI) [25] , Empty Test (EM) [25], Exception Handling (EH) [25], Redundant Print (RP) [25], Redundant Assertion (RA) [25], Sensitive Equality (SE) [24], Sleepy Test (ST) [25], Eager Test (EA) [24], Lazy Test (LT) [24], Duplicate Assert (DA) [25], Unknown Test (UR) [25], Ignored Test (IT) [25], and Magic Number Test (MNT) [26]. 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.

Whereas Codex and ChatGPT-3.5 did not produce unit tests with the Exception of Handling (EH) smell, this smell type was frequent in all test cases manually created, and the ones generated by Evosuite. **EH** smell also ocurred in the 7.6% of the (compilable) unit tests generated by CodeGen. This smell occurs when the test method itself captures exceptions to pass/fail a test instead of using the `expected` attribute from the `@Test` annotation.

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).

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

TABLE IV
TEST SMELLS DISTRIBUTION FOR THE HUMANEVAL DATASET (RQ1).

Smell	Codex (2K)	Codex (4K)	CodeGen	ChatGPT-3.5	Evosuite	Manual
AR	61.3%	59.7%	41.8%	23.8%	15.0%	0.0%
CLT	0.0%	0.0%	6.2%	1.5%	0.0%	0.0%
CI	0.0%	0.0%	10.2%	0.0%	0.0%	0.0%
EM	1.9%	1.3%	7.4%	0.8%	0.0%	0.0%
EH	0.0%	0.0%	7.6%	0.0%	100.0%	100.0%
RP	0.0%	0.0%	7.6%	0.0%	0.0%	0.0%
RA	0.0%	0.0%	1.5%	0.0%	0.0%	0.0%
SE	0.0%	0.0%	1.9%	0.0%	0.0%	0.0%
ST	0.0%	0.0%	0.2%	0.0%	0.0%	0.0%
EA	60.6%	59.1%	41.4%	23.8%	16.3%	0.0%
LT	39.4%	41.5%	17.6%	86.2%	99.4%	100.0%
DA	15.6%	14.5%	13.0%	3.1%	0.6%	0.0%
UT	10.0%	5.7%	14.9%	0.8%	0.0%	0.0%
IT	0.0%	0.0%	0.2%	0.0%	0.0%	0.0%
MNT	100.0%	100.0%	91.3%	100.0%	100.0%	100.0%

2) Results for SF110 dataset: The smells detected for the SF110 tests are listed in Table V. Similar to HumanEval, Magic Number Test smell, Assertion Roulette, and Eager Tests are frequently reoccurring smells on the LLMs and the Evosuite. Unlike Evosuite, LLMs produced more Empty Tests (EM) (28.7%) and Constructor Initialization (CI) (9.3%) smells.

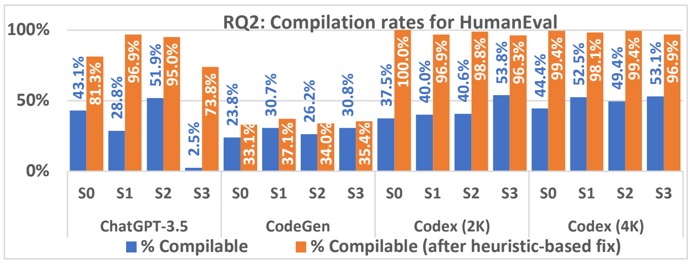
TABLE V
TEST SMELLS DISTRIBUTION FOR THE SF110 DATASET (RQ1).

Smell	Codex (2K)	Codex (4K)	CodeGen	ChatGPT-3.5	Evosuite
AR	14.4%	17.1%	3.6%	4.6%	35.0%
CLT	0.5%	1.3%	1.4%	2.3%	0.0%
CI	0.0%	0.7%	8.6%	0.0%	0.1%
EM	7.2%	1.3%	20.1%	0.0%	0.0%
EH	20.7%	19.1%	13.7%	2.3%	91.2%
MG	2.7%	3.3%	0.7%	0.0%	3.0%
RP	4.5%	5.9%	5.0%	0.0%	0.0%
RA	0.9%	0.7%	0.7%	0.0%	0.0%
SE	0.9%	1.3%	2.2%	0.0%	13.7%
EA	28.4%	31.6%	6.5%	12.6%	39.6%
LT	60.8%	60.5%	1.4%	21.8%	46.4%
DA	1.4%	2.0%	0.0%	1.1%	1.5%
UT	21.2%	10.5%	25.9%	0.0%	22.9%
RO	2.7%	3.9%	0.7%	0.0%	2.7%
MNT	93.2%	96.1%	50.4%	21.8%	91.2%

V. RQ2 RESULTS: CODE ELEMENTS IN A CONTEXT

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*.

A. Compilation Status



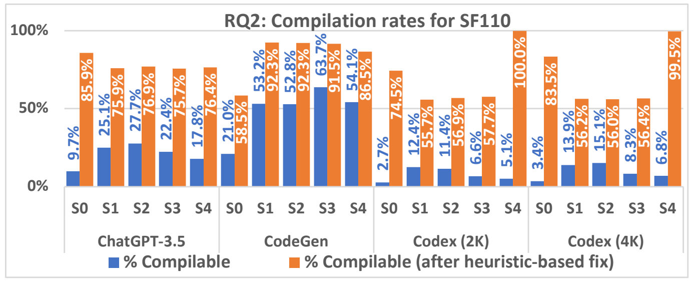


Fig. 1. Compilation rates for HumanEval and SF110 across different scenarios

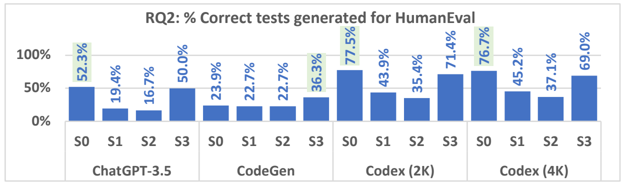
1) HumanEval Results: Fig. 1 shows the compilation rates for the HumanEval dataset across different scenarios and LLMs. We observed that the scenario 3 increased the original (S0) compilation rates for CodeGen, and Codex (2K and 4K) from 23.8%, 37.5%, and 44.4% to **30.8%**, **53.8%** and **53.1%**, respectively. Although scenario 3 increased the original compilation rates (blue bars in Fig. 1) these tests have similar heuristic-based fix rates (except for CodeGen). In the case of CodeGen, S3 achieved a final compilation rate after applying heuristics equal to **35.4%** which is **2%** higher than fixing the outputs generated from the original prompt (**33.1%**).

ChatGPT-3.5, 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 ChatGPT 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 placed in the very first line (corresponding to the CUT's package) and the other 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**.

2) SF110 Results: S2 increased the original (S0) compilation rates for ChatGPT3.5 and Codex (4K), as shown in Fig. 1. However, scenarios 3 and 1 were the best performers for CodeGen, and Codex (2K), respectively. Hence, no scenario achieved a consistent best performance overall.

B. Test Correctness

1) HumanEval Results: Fig. 2 depicts the percentage of unit tests generated by the models that are *correct*. Among all scenarios, scenario 3 had a similar correctness rate compared to the original prompt used in RQ1 for ChatGPT-3.5 and Codex (2K, 4K). In case of CodeGen, it produced **11.7%** more correct tests (**35.9%**). It is important to highlight that whereas ChatGPT-3.5 only had 73.8% compilable tests in scenario3 (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.



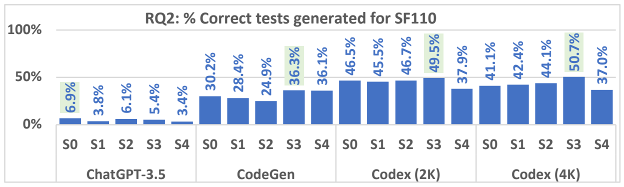
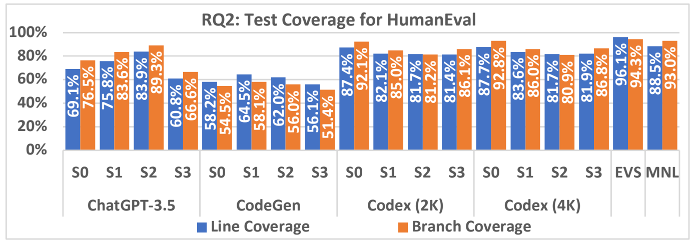


Fig. 2. Correctness rates across different datasets, scenarios, and LLMs

2) SF110 Results: As shown in Fig. 2, while the original prompt achieved the highest correctness rate for ChatGPT-3.5 (6.9%), the other LLMs observed a correctness increase when using the context from scenario 3. Codex (4K) experienced the highest increase (from 37.9% to 50.7%).

C. Test Coverage



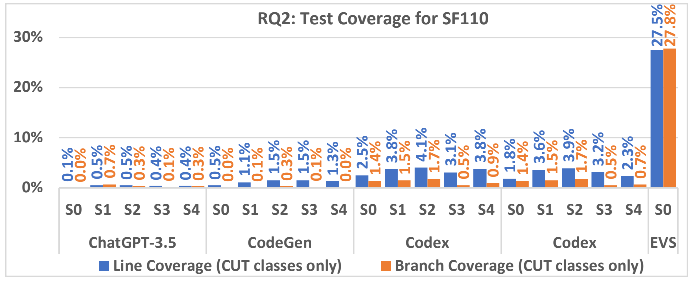


Fig. 3. Line and Branch Coverage across different datasets, scenarios, and LLMs (EVS = Evosuite; MNL = Manual).

1) HumanEval Results: The observed line and branch coverage for each scenario and LLMs is shown in Fig. 3, which shows there is no clear trend in terms of consistent scenario performance. In the case of Codex and CodeGen, scenario 1 is the one that had the highest line coverage among the different scenarios in these models. ChatGPT3.5, 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, scenario 1 was the best one for CodeGen, and scenario 2 is the best one for ChatGPT-3.5. None of the scenarios in Codex outperformed the line/branch coverage of the original prompts nor the coverage achieved by the manual and Evosuite's tests.

2) SF110 Results: Among all scenarios, scenario 2 had a slightly higher line coverage (0.4%–2.1% increase) when compared to the original prompt used in RQ1 for all LLMs. In the case of branch coverage, S1 had slightly higher coverage for ChatGPT-3.5, whereas S2 was the best one for the remaining LLMs. However, these increases are still much lower than Evosuite's test coverage, which achieved ≈ 27% line and branch coverage.

TABLE VII
TEST SMELLS DISTRIBUTION FOR THE SF110 DATASET (RQ2).

	Codex (2K)				Codex (4K)				CodeGen				ChatGPT-3.5			
	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%	3.8%	6.7%	6.4%	6.1%	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%	0.7%	1.9%	2.3%	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%	6.9%	10.5%	12.1%	15.2%	0.0%	0.0%	0.0%	0.0%
DT	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	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%	16.3%	16.1%	19.0%	19.3%	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%	10.5%	12.3%	16.2%	19.3%	2.2%	3.3%	2.7%	5.0%
GF	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
MG	2.0%	1.5%	1.0%	2.6%	1.0%	1.5%	1.5%	2.5%	0.7%	0.2%	0.5%	3.0%	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%	3.1%	3.4%	5.9%	6.1%	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%	0.9%	1.1%	1.8%	1.9%	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%	0.2%	0.8%	2.1%	0.0%	0.5%	0.6%	1.1%	2.1%
VT	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
ST	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%
EA	16.8%	14.4%	11.4%	20.8%	17.0%	13.0%	11.6%	25.3%	3.6%	6.3%	5.9%	6.8%	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%	2.0%	2.3%	2.3%	1.9%	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%	0.7%	2.3%	0.8%	1.5%	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%	26.5%	25.9%	31.0%	33.3%	0.0%	0.0%	1.6%	2.1%
IT	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	0.4%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%
RO	1.5%	1.5%	2.0%	2.2%	1.0%	1.5%	2.5%	2.9%	0.7%	0.6%	0.5%	3.0%	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%	47.0%	49.0%	66.7%	63.3%	18.6%	21.1%	18.0%	29.1%
DPT	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

D. Test Smells

1) HumanEval Results: Table VI 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 IV). Moreover, 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 ChatGPT-3.5 and CodeGen 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 VI
TEST SMELLS DISTRIBUTION FOR THE HUMANEVAL DATASET (RQ2).

	ChatGPT-3.5			CodeGen			Codex (2K)			Codex (4K)		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
AR	7.1%	11.8%	30.5%	39.7%	41.5%	34.9%	16.8%	38.6%	61.0%	16.6%	40.3%	63.2%
CLT	6.5%	3.3%	0.8%	4.4%	5.1%	3.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
CI	0.0%	0.0%	0.0%	5.6%	6.8%	10.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
EM	0.0%	0.7%	3.4%	6.1%	5.1%	21.5%	4.5%	3.2%	1.9%	1.3%	1.3%	1.9%
EH	0.0%	0.0%	0.0%	5.7%	5.9%	9.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
RP	0.0%	0.0%	0.0%	5.2%	5.5%	5.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
RA	0.0%	0.0%	0.0%	3.4%	1.8%	2.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
SE	0.0%	0.0%	0.0%	0.2%	2.8%	2.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
ST	0.0%	0.0%	0.0%	0.0%	0.0%	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
EA	7.1%	10.5%	26.3%	40.1%	41.7%	34.0%	15.5%	37.3%	56.5%	15.3%	38.4%	58.1%
LT	85.2%	92.8%	82.2%	13.0%	17.8%	13.4%	84.5%	60.8%	44.2%	84.7%	60.4%	42.6%
DA	1.3%	0.0%	1.7%	6.6%	8.8%	7.8%	0.6%	8.2%	11.0%	1.9%	6.9%	11.6%
UT	0.0%	0.7%	3.4%	13.6%	12.7%	30.0%	13.5%	16.5%	2.6%	5.1%	8.2%	2.6%
IT	0.0%	0.0%	0.0%	0.2%	0.0%	0.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
MNT	89.7%	98.7%	100.0%	95.3%	95.4%	91.5%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%

2) SF110 Results: As shown Table VII, there is not any scenario that consistently outperforms the other. However, we can observe that CodeGen and ChatGPT produce more test smells across the scenarios, as we can see from the cells highlighted in red.

VI. RESULTS SUMMARY AND IMPLICATIONS

In this section, we summarize the findings and provide the implications of each of them.

– **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, as we showed in Table I. For example, while Evosuite produced one unit test for each of the 160 classes under test, ChatGPT3.5 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 ChatGPT generated test cases for "stress-testing", such as using `Integer.MAX_VALUE` and similar inputs in order to test for the MUT's behavior in the face of exceptionally large inputs.

– **CodeGen and Codex are the least and best performing LLMs, respectively**. This can be explained by the fact that CodeGen is a smaller model with 350 million parameters trained with 341.1 Gb of data, compared to Codex and ChatGPT-3.5, which has 12 billion and 175 billion parameters, respectively and trained with over 40 terabytes of data. Moreover, besides being a more powerful model, Codex is an LLM fine-tuned for code-related tasks in contrast to ChatGPT, 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 default classpath.

– **LLMs may not reason over path feasibility**. We observed 3 unit tests from CodeGen getting stuck in an infinite loop. We also found that when a method's static return type is a supertype (*e.g.*, `Set`), but its actual implementation only returns two specific runtime types (*e.g.*, `HashSet` and `TreeSet`), the LLMs were often unable to generate expected outputs whose types are only the actual feasible runtime types. Besides, we found that LLMs do not understand type erasure and Java Generics. LLMs attempted to assign arrays to lists (and vice-versa) or used incorrect types that violate the upper/lower bounds of the generic type.

– **Synergy between LLMs and TDD**. Although LLMs were not capable of achieving coverages or compilation rates comparable to Evosuite, the LLMs 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 [2] 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.

A. Threats to Validity

Creating canonical solutions for the Java samples in the HumanEval dataset [17] introduced an internal validity threat. To mitigate it, we extensively vetted our solution with a test set provided by the dataset creator; they passed the test cases without any problem. Another validity threat relates to the use of SF110 benchmark [13], JaCoCo [42] for calculating coverage results and TsDetect [45] for finding test smells. In this case, our analyses depend on the representativeness of the SF110 dataset (construct validity threat) and accuracy of these tools. However, the SF110 dataset is commonly used to benchmark automated test generation tools [13], [47], [48] and the used tools are well-known among researchers and practitioners [49], [50].

VII. Related Work

Previous works have focused on creating source code that can do a specific task automatically (code generation). The deductive synthesis approach [51], [52], 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 [53]. Recurrent networks were used by Yin *et al.* [54] 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 [35], CodeGen [16] and CodeT5 [37]) after being refined on enormous code datasets. Later, GitHub Copilot developed an improved auto-complete mechanism using the upgraded version of Codex [14], which can help to solve fundamental algorithmic problems [4]. Recent works [55]–[57] focus on optimizing the process to create, fine-tune, and infer the Large Language Models-based code generation techniques. Using large language models for software test generation is not that common. However, they can be used for downstream tasks, for example, flaky test prediction [58]. However, recent work uses GPT-3 [31] for software graphical interface testing [59]. 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.* [11] 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.* [60] 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.* [61] presented a technique that improves random test generation by incorporating feedback obtained from executing test inputs as they are created for generating unit tests. Pecorelli *et al.* [62] conducted an empirical study on software testing for Android applications about finding effectiveness, design, and bugs in the production code. 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.* [63] used Codex [14] to automatically generate unit tests using an adaptive approach. They used 25 npm packages to evaluate their tool, TESTPIOLOT. 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.* [64] 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.* [65] 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. Bareiß *et al.* [66] 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.* [65]. 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.

VIII. Conclusion

We investigated the capability of three code generation models for unit test generation. We conducted experiments with different contexts in the prompt and compared the result 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] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[2] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: ACM, 2022, p. 21–29. [Online]. Available: https://doi.org/10.1145/3520312.3534864

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[4] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming *et al.*, "Github copilot ai pair programmer: Asset or liability?" *arXiv preprint arXiv:2206.15331*, 2022.

[5] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *2022 IEEE 22nd Int'l Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2022, pp. 71–82.

[6] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 980–994. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00057

[7] M. A. Hadi, I. N. B. Yusuf, F. Thung, K. G. Luong, J. Lingxiao, F. H. Fard, and D. Lo, "On the effectiveness of pretrained models for api learning," in *Proceedings of the 30th IEEE/ACM Int'l Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: ACM, 2022, p. 309–320. [Online]. Available: https://doi.org/10.1145/3524610.3527886

[8] M. L. Siddiq, A. Samee, S. R. Azgor, M. A. Haider, S. I. Sawraz, and J. C. Santos, "Zero-shot prompting for code complexity prediction using github copilot," in *2023 The 2nd Intl. Workshop on NL-based Software Engineering*, 2023.

[9] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *2012 7th Int'l Workshop on Automation of Software Test (AST)*. IEEE, 2012, pp. 36–42.

[10] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[11] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser, "How do automatically generated unit tests influence software maintenance?" in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 250–261.

[12] L. Moonen, A. v. Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software evolution*. Springer, 2008, pp. 173–202.

[13] 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

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto *et al.*, "Evaluating large language models trained on code," 2021.

[15] "Chat completions," Accessed Mar 25, 2023, 2023. [Online]. Available: https://platform.openai.com/docs/guides/chat

[16] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint*, 2022.

[17] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li *et al.*, "Multi-lingual evaluation of code generation models," 2022. [Online]. Available: https://arxiv.org/abs/2210.14868

[18] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 2012, pp. 178–188.

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

[20] D. Gonzalez, J. C. Santos, A. Popovich, M. Mirakhorli, and M. Nagappan, "A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 391–401.

[21] E. M. Guerra and C. T. Fernandes, "Refactoring test code safely," in *International Conference on Software Engineering Advances (ICSEA 2007)*, 2007, pp. 44–44.

[22] M. Greiler, A. Zaidman, A. v. Deursen, and M.-A. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, p. 387–396.

[23] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "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)*, 2016, pp. 5–14.

[24] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings 2nd Int'l Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, M. Marchesi and G. Succi, Eds., may 2001, Conference.

[25] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual Int'l Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2019, p. 193–202.

[26] G. Meszaros, S. M. Smith, and J. Andrea, "The test automation manifesto," in *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, F. Maurer and D. Wells, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 73–81.

[27] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf

[28] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, mar 2020. [Online]. Available: https://doi.org/10.1016%2Fj.physd.2019.132306

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[30] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[31] T. Brown, B. Mann, N. Ryder, M. Subbiah *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[32] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *44th Int'l Conference on Software Engineering (ICSE)*, 2022.

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

[34] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th Int'l Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 329–340.

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[36] Y. Gao and C. Lyu, "M2ts: Multi-scale multi-modal approach based on transformer for source code summarization," *arXiv preprint arXiv:2203.09707*, 2022.

[37] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[38] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: Enhancing continuous integration with automated test generation," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 55–66.

[39] G. Fraser and A. Arcuri, "A large scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[40] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *CoRR*, vol. abs/2204.05999, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.05999

[41] N. Coooper, A. Arutiunian, S. Hincapié-Potes, B. Trevett, A. Raja, E. Hossami, M. Mathur *et al.*, "GPT Code Clippy: The Open Source version of GitHub Copilot," Jul. 2021. [Online]. Available: https://github.com/CodedotAl/gpt-code-clippy/wiki

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

[43] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, p. 955–963. [Online]. Available: https://doi.org/10.1145/3338906.3340459

[44] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, p. 53–63. [Online]. Available: https://doi.org/10.1145/3238147.3238183

[45] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "TsDetect: An open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1650–1654. [Online]. Available: https://doi.org/10.1145/3368089.3417921

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

[47] D. Bruce, H. D. Menéndez, and D. Clark, "Dorylus: An ant colony based tool for automated test case generation," in *Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Tallinn, Estonia, August 31–September 1, 2019, Proceedings 11*. Springer, 2019, pp. 171–180.

[48] M. M. D. Shahabi, S. P. Badiei, S. E. Beheshtian, R. Akbari, and S. M. R. Moosavi, "On the performance of evopso: A pso based algorithm for test data generation in evosuite," in *2017 2nd Conference on Swarm Intelligence and Evolutionary Computation (CSIEC)*. IEEE, 2017, pp. 129–134.

[49] I. Bilal, I. Al-Taharwa, S. Rami, I. M. Alkhawaldeh, and N. Ghatasheh, "Jacoco-coverage based statistical approach for ranking and selecting key classes in object-oriented software," *J. Eng. Sci. Technol*, vol. 16, pp. 3358–3386, 2021.

[50] T. Virgínio, L. Martins, R. Santana, A. Cruz, L. Rocha, H. Costa, and I. Machado, "On the test smells detection: an empirical study on the jnose test accuracy," *Journal of Software Engineering Research and Development*, vol. 9, pp. 8–1, 2021.

[51] C. Green, "Application of theorem proving to problem solving," in *Proceedings of the 1st Int'l Joint Conference on Artificial Intelligence*, ser. IJCAI'69. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1969, p. 219–239.

[52] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Commun. ACM*, vol. 14, no. 3, p. 151–165, mar 1971. [Online]. Available: https://doi.org/10.1145/362566.362568

[53] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[54] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 440–450. [Online]. Available: https://aclanthology.org/P17-1041

[55] A. Madaan, S. Zhou, U. Alon, Y. Yang, and G. Neubig, "Language models of code are few-shot commonsense learners," 2022. [Online]. Available: https://arxiv.org/abs/2210.07128

[56] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[57] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *arXiv preprint arXiv:2207.01780*, 2022.

[58] S. Fatima, T. A. Ghaleb, and L. Briand, “Flakify: A black-box, language model-based predictor for flaky tests,” *IEEE Transactions on Software Engineering*, 2022.

[59] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, “Fill in the blank: Context-aware automated text input generation for mobile gui testing,” 2022. [Online]. Available: https://arxiv.org/abs/2212.04732

[60] 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.

[61] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *29th International Conference on Software Engineering (ICSE’07)*. IEEE, 2007, pp. 75–84.

[62] F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba, “Software testing and android applications: a large-scale empirical study,” *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–41, 2022.

[63] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “Adaptive test generation using a large language model,” *arXiv preprint arXiv:2302.06527*, 2023.

[64] 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 International Conference on Software Engineering, ser. ICSE*, 2023.

[65] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” *ICSE23*, 2023.

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