Information and Software Technology 171 (2024) 107468

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Effective test generation using pre-trained Large Language Models and mutation testing

Arghavan Moradi Dakhel *, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, Michel C. Desmarais

Department of Computer and Software Engineering, Polytechnique Montreal, Montreal, H3T 1J4, Quebec, Canada

A R T I C L E I N F O

Dataset link: https://github.com/ExpertiseModel/MuTAP

Keywords:
Test generation
Large language model
Mutation testing

A B S T R A C T

Context: One of the critical phases in the software development life cycle is software testing. Testing helps with identifying potential bugs and reducing maintenance costs. The goal of automated test generation tools is to ease the development of tests by suggesting efficient bug-revealing tests. Recently, researchers have leveraged Large Language Models (LLMs) of code to generate unit tests. While the code coverage of generated tests was usually assessed, the literature has acknowledged that the coverage is weakly correlated with the efficiency of tests in bug detection.
Objective: To improve over this limitation, in this paper, we introduce *MuTAP* (**Mu**tation **T**est case generation using **A**ugmented **P**rompt) for improving the effectiveness of test cases generated by LLMs in terms of revealing bugs by leveraging mutation testing.
Methods: Our goal is achieved by augmenting prompts with surviving mutants, as those mutants highlight the limitations of test cases in detecting bugs. *MuTAP* is capable of generating effective test cases in the absence of natural language descriptions of the Program Under Test (PUTs). We employ different LLMs within *MuTAP* and evaluate their performance on different benchmarks.
Results: Our results show that our proposed method is able to detect up to 28% more faulty human-written code snippets. Among these, 17% remained undetected by both the current state-of-the-art fully-automated test generation tool (i.e., Pynguin) and zero-shot/few-shot learning approaches on LLMs. Furthermore, *MuTAP* achieves a Mutation Score (MS) of 93.57% on synthetic buggy code, outperforming all other approaches in our evaluation.
Conclusion: Our findings suggest that although LLMs can serve as a useful tool to generate test cases, they require specific post-processing steps to enhance the effectiveness of the generated test cases which may suffer from syntactic or functional errors and may be ineffective in detecting certain types of bugs and testing corner cases in *PUT*s.

1. Introduction

Testing is an important yet expensive step in the software development lifecycle. Generating effective tests is a time-consuming and tedious task for developers. Unit tests are essential as they form the basis of the test automation pyramid [1,2]. Unit tests check if a function or a component works as expected in isolation. A unit test consists of two components: the first component is a set of test inputs for the Program Under Test (*PUT*), while the second component is the test oracle that indicates the intended behavior (output) of the *PUT* and is, therefore, capable of exposing bugs by verifying the correctness of the *PUT* on test inputs [3]. A test oracle can be in the format of assertions.

The automatic generation of unit tests is an important topic in Software Engineering (SE). It aims to reduce developers' testing efforts. Developing good-quality unit tests can prevent bugs in software products. There are different tools for automatically generating unit tests and test suites that are either based on random test generators [4,5], dynamic symbolic execution [6,7], or search-based approaches [8,9]. However, these techniques have some drawbacks and often generate tests with no assertion or too general assertions, or tests with assertions that cannot effectively assess the intended behavior of the *PUT* [10,11].

Considering these shortcomings, researchers have recently been exploring the possibility of leveraging Machine Learning-based code

* Corresponding author.
E-mail addresses: arghavan.moradi-dakhel@polymtl.ca (A.M. Dakhel), amin.nikanjam@polymtl.ca (A. Nikanjam), vahid.majdinasab@polymtl.ca (V. Majdinasab), foutse.khomh@polymtl.ca (F. Khomh), michel.desmarais@polymtl.ca (M.C. Desmarais).

https://doi.org/10.1016/j.infsof.2024.107468
Received 31 August 2023; Received in revised form 2 April 2024; Accepted 3 April 2024
Available online 6 April 2024

synthesis techniques for generating better unit tests [12–16]. Specifically, these approaches have been exploring the potential of Large Language Models (LLMs) with the transformer architecture, such as Codex [17], which has achieved good performance in automatic program synthesis [17–21]. Among such efforts, Bareißet al. [12] evaluate Codex's performance for test case generation by using a *few-shot* learning approach. Their findings on a limited set of 18 Java methods show that their approach is comparable to feedback-directed test generation. ATHENATEST [22] leveraged the BART transformer model [23] after fine-tuning it on a set of real Java functions and their corresponding tests. They also reported achieving comparable coverage to EvoSuite [9] after an assessment of five Java projects. Lemieux et al. [16] proposed CODAMOSA which utilized test cases generated by Codex to improve search-based testing techniques, which consists of only the prefix (inputs) of a test case without any test oracles. Their reported results obtained on 27 Python projects show that CODAMOSA surpasses the baseline search-based technique, Pynguin [24] and Codex in terms of code coverage. Although the preliminary results of these studies and others [13–15,25], are promising, none of these studies attempted to improve the bug detection capability of generated tests. Moreover, it has been acknowledged in the literature that while test coverage is a useful metric for evaluating the quality of tests, it is weakly correlated with the efficiency of tests in bug detection [26–28].

Mutation Testing (MT) is a white box testing technique to assess the capability of a test in revealing bugs. MT has been widely studied and successfully used in SE to assess the effectiveness of test cases [29,30]. MT involves injecting *artificial* changes based on *real* faults into a *PUT*, resulting in mutated versions of the *PUT* known as mutants. The more a test case kills mutants, the more effective it is in identifying real bugs. The surviving mutants highlight the weaknesses of a test case and the ultimate goal is for the test cases to be able to detect all mutants, i.e., kill them. Mutants are not only useful for assessing the effectiveness of test cases but can also be used as a means for designing more effective test cases [9].

In this paper, we present the first study that leverages MT to enhance and evaluate the effectiveness of test cases generated by LLMs for Python programs in terms of fault revealing capabilities. Our approach aims to optimize test cases for bug detection rather than code coverage. Our proposed technique, *MuTAP*, employs an LLM as its main Component (LLMC) and starts by feeding a prompt to the LLMC in order to generate test cases. The initial prompt includes the *PUT* and instructions for generating test cases by using *zero-shot* and *few-shot* learning. Next, *MuTAP* assesses the syntax of the generated test cases and re-prompts its LLMC to rectify any detected syntax issues. After fixing syntax errors, *MuTAP* proceeds to appraise the intended behavior of the generated test cases. This is achieved by comparing the output of the test oracles on certain test inputs to the expected return values of the PUT using the same test inputs, thereby correcting any unintended behavior in the test oracles.

Subsequently, *MuTAP* applies MT to examine the effectiveness of test cases in killing mutants of *PUT*s. As surviving mutants highlight the limitation of the generated test cases, *MuTAP* re-prompts its LLMC to generate new test cases for the *PUT*s that have surviving mutants by augmenting the initial prompt with both initial test cases and the surviving mutants. *MuTAP* halts the process of augmenting the initial prompt when either the final test cases can effectively detect all mutants or there are no surviving mutants left that have not already been used to augment the initial prompt.

We employ two types of LLMs as the LLMC of *MuTAP*: *Codex*, which is designed for code-related tasks, and *llama-2-chat*, which is optimized for dialog use cases and versatile enough to accommodate a range of tasks, including programming. We evaluate *MuTAP* on both synthetic bugs of 164 *PUT*s [17] and 1710 buggy programs collected from a Python bug repairing benchmark [31].

Our results indicate that our proposed approach generates effective test cases with an average Mutation Score (MS, the ratio of killed mutants by the total number of mutants) of 93.57%, outperforming both Pynguin (a state-of-the-art fully-automated test generation tool) and the conventional LLM-based zero-shot/few-shot learning techniques. Furthermore, our approach detects up to 468 (28%) more buggy code snippets written by humans than other comparable methods in our evaluation. Remarkably, it identifies 79 (17%) buggy code snippets of humans that none of the other techniques are able to detect. To summarize, this paper makes the following contributions:

- We present the first study on leveraging MT to generate test cases with LLMs.
- We propose a prompt-based learning technique to improve the effectiveness of test cases by augmenting the prompts with both initial test cases and surviving mutants of a *PUT*.
- We assess the effectiveness of generated tests in detecting bugs in real and synthetic buggy versions of *PUT*s.
- We make the proposed technique, *MuTAP*, publicly available online [32] for other researchers/practitioners to replicate or build upon our work.

The rest of this paper is organized as follows. Section 2 introduces a motivating example. Section 3 describes the different steps of our approach. We present our experimental setup, research questions, and experimental results in Section 4. We discuss our findings and the potential use cases of our approach in Section 5. Threats to the validity of our results are reviewed in Section 6. We briefly review the related works in Section 7. Finally, we conclude the paper in Section 8; highlighting some avenues for future works.

2. Motivating example

In this section, we present an example in Fig. 1 showing how our proposed approach generates effective test cases. Suppose we have 10 mutants $\{SM_0, SM_1, \dots, SM_9\}$ for the Program Under Test, PUT in Fig. 1. The PUT is a function that takes a certain expected input and produces a desired output upon applying its functionality. The goal of our proposed technique, *MuTAP* (**Mu**tation **T**est case generation using **A**ugmented **P**rompt), is to generate effective test cases for PUT in a way that ensures killing the maximum number of mutants.

The function *any_int()* in Fig. 1 receives 3 inputs and returns *True* if all 3 inputs are integers, also one of the inputs is equal to the sum of two others. Otherwise, it returns *False*. In the first step, *MuTAP* uses the initial prompt, 1, to run a query on the LLM Component (LLMC) and generates initial test cases for this Program Under Test (PUT). The component 2 in Fig. 1 shows the initial test cases generated by LLMC after the refining step. We named it Initial Unit Test, *IUT*. In Section 3, we discuss the refining step (syntax and intended behavior fixing) of our approach in detail. The *IUT* kills 6 out of 10 mutants of *PUT*. The 4 remaining mutants reveal the weaknesses of the generated test, meaning that *IUT* needs new test cases with assertion to kill the injected bugs in those 4 mutants.

To address this limitation and generate more effective test cases, *MuTAP* augments the initial prompt with two new components; the first one is the response of the model to the initial prompt after fixing its syntax and intended behavior, *IUT*, and the second one is the mutant component, 3 in Fig. 1. *MuTAP* initiates the construction of the mutant component by using the first "Survived Mutant" of *PUT* that we refer to as SM_0. The red highlight in SM_0 shows the injected bug in PUT. The injected bug changes the second statement in the condition of the inner *if* in PUT in a way that the sum of the first and last input of function *any_int()* is not equal to the middle input anymore. Since there is no test case in *IUT* to verify that its middle input, y, is equal to the sum of its first and last inputs, x and z, *IUT* is not able to kill this mutant.

MuTAP uses the concatenation of these three components: 1, 2, and 3 to re-prompt the LLMC. The 4 component in Fig. 1, shows the new set of test cases generated by LLMC appended to *IUT* after the refining step. We named it Augmented Unit Test, AUT_0. The unit

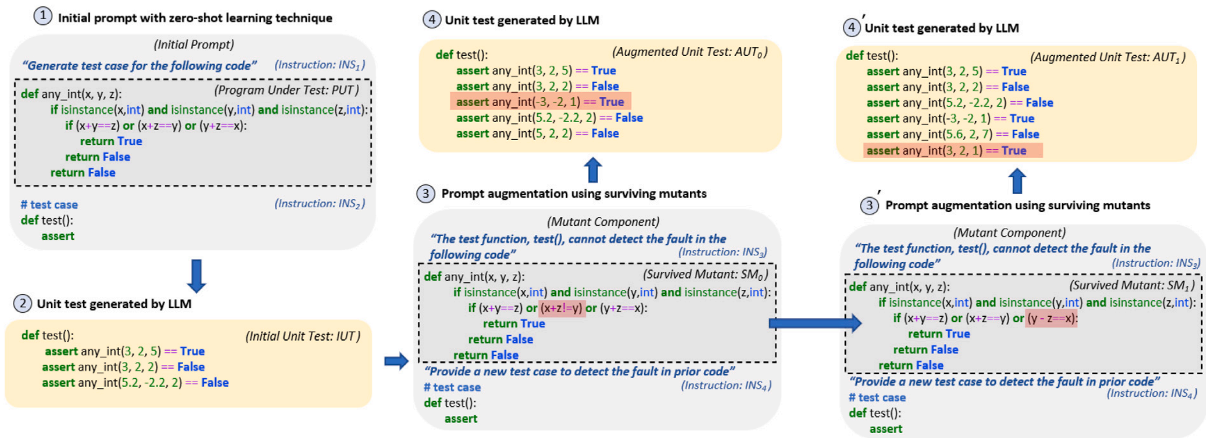

Fig. 1. Different steps of *MuTAP* on a PUT. ② is a set of test cases generated by the initial prompt ① for *PUT*, and ④ is a set of test cases obtained after augmenting the initial prompt with the surviving mutant, SM_0. ③′ shows the mutant component after updating with another surviving mutant of PUT_0 that we named SM_1.

test has two more assertions compared to the *IUT* and one of them, highlighted in red, kills the mutant, SM_0.

MuTAP applied AUT_0 to the mutants of *PUT* again. If there are any remaining surviving mutants, *MuTAP* iterates the augmentation process by updating the mutant component with another surviving mutant if it has not been used to augment the prompt previously. *MuTAP* utilizes each mutant individually because sometimes new test cases that address one mutant can also kill the remaining surviving mutants. Moreover, due to the limited length of the prompt and non-constant length of mutants, applying each surviving mutant separately is a more practical approach. Fig. 1 ③′ shows an example of how the mutant component is updated using another surviving mutant. We call this mutant SM_1. Unit test, ④′, shows a new set of test cases including one assertion that detects SM_1. *MuTAP* iterates the augmentation process until either the final test cases can kill all the mutants, or there are no surviving mutants left that have not already been used to augment the initial prompt.

The final test cases generated by our proposed technique, *MuTAP*, kill 9 out of 10 mutants of this example, *PUT*, and it increases the MS for *PUT* from 60% (6 out of 10) to 90% (9 out of 10). This result can be compared to the state-of-the-art automatic test generation tool for Python programming language [24], Pynguin, which generates a test case for *PUT* with only a 40% MS. This tool uses a search-based generation technique [33] and randomly mutates the test values within a test case to generate new test cases. The random nature of this method results in a low chance of generating a new test case that can kill the surviving mutants of *PUT*.

3. Approach

In this section, we discuss the different steps of our approach. Fig. 2 shows an overview of our proposed approach and Algorithm 1 presents the sequence of its different steps. MuTAP initiates the process by invoking an initial prompt on its LLMC to generate test cases for a specific PUT. Subsequently, it applies refining steps to repair the syntax and intended behavior of the generated test cases. Once the test cases are corrected, MuTAP proceeds to the MT step by generating various mutants for the PUT and calculating the MS. If there are any surviving mutants leading to MS < 100%, MuTAP uses those surviving mutants in different iterations to augment the initial prompt. It then re-prompts its LLMC with the augmented prompt in each iteration of the augmentation step to generate new test cases. At the end of each iteration, MuTAP recalculates the MS on the same set of mutants to determine if the new test cases eliminate all surviving mutants or if there are remaining surviving mutants, prompting further iterations. If MS=100% or all the surviving mutants are already incorporated into the augmentation step, MuTAP proceeds to the Oracle Minimization step to eliminate redundant test cases.

Algorithm 1: *MuTAP*

```
Input: PUT, LLMC, initial_prompt_type
/* INS1, INS2, INS3, INS4 and INSfix are global variable as
   natural language instructions for the prompts                 */
Output: FUT // Final Unit Test
// Initial Prompt
initial_prompt ← GenerateInitialPrompt (PUT, initial_prompt_type)
raw_IUT ← LLMC (initial_prompt)
// Syntax Fixer and Intended Behavior Repair
IUT ← Refining (raw_IUT, PUT)
// Mutation Testing
mutants ← MutPy(PUT)
MS, surviving_mutant ← MutationTesting (IUT, mutants)
while MS < 100% or surviving_mutant ≠ {} do
    SM ← surviving_mutant.pop()
    // Prompt Augmentation
    augmented_prompt ← AugmentingPrompt (initial_prompt, IUT, SM)
    raw_AUT ← LLMC (augmented_prompt)
    fixed_AUT ← Refining (raw_AUT, PUT)
    IUT ← IUT.append(fixed_AUT)
    MS , augmnt_surviving_mutant ← MutationTesting (IUT, mutants)
    surviving_mutant ← surviving_mutant ∩ augmnt_surviving_mutant
end
// F: Oracle Minimization
FUT ← OracleMinimization (IUT, mutants)
return FUT
```

3.1. Initial prompt

LLMs are capable of performing those tasks that they are already trained for. Fine-tuning LLMs to perform a new task is computationally expensive. Also, there are LLMs such as Codex that show a very good performance in generating code but since they are closed-source, fine-tuning them for a new task is impossible.

Prompt-based learning [34,35] is an effective technique to adapt LLMs for new tasks. A prompt is a combination of natural language and/or programming language context and is used as an input to LLMs. There are studies showing that putting a natural language instruction as a hint (*zero-shot learning*) [15,16,36] or several examples (*few-shot learning*) [37–39] in the prompt increases the capability of LLMs in performing a new task.

MuTAP employs both *zero-shot* and *few-shot* learning to build the initial prompt and calls LLMC on them separately. This step is shown in Algorithm 2. In more detail, we employ *zero-shot* and *few-shot* as follows:

- *zero-shot*: The initial prompt generated by *zero-shot* technique contains three units [16]. The component indicated by ① in Fig. 1 shows an example of such a prompt. The first unit in

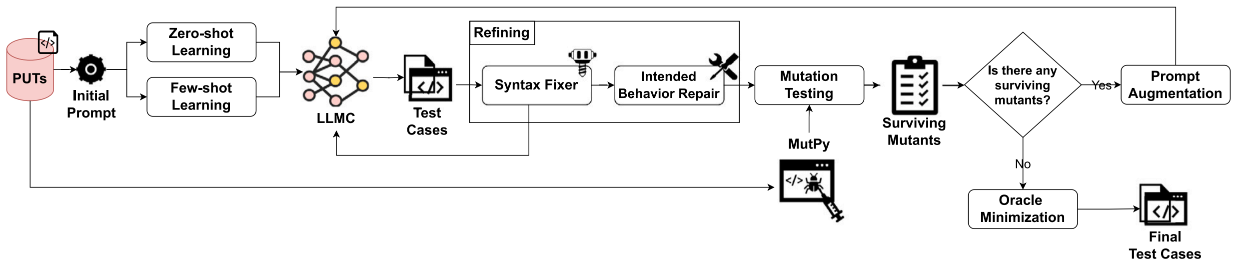

Fig. 2. The proposed methodology for generating and evaluating tests using LLMs.

Algorithm 2: GenerateInitialPrompt

Input: *PUT, initial_prompt_type*
Output: *initial_prompt*
1 **if** *initial_prompt_type* == *"zero-shot"* **then**
2   *initial_prompt* ← CONCAT(INS_1, *PUT*, INS_2)
3 **else**
4   **if** *initial_prompt_type* == *"few-shot"* **then**
5   *initial_prompt* ← CONCAT(pair(*M*,*UT*), *PUT*) `// M: Method, UT: Unit Test`
6   **end**
7 **end**
8 **return** *initial_prompt*

this component is an instruction in a natural language named INS_1 and it clarifies the task by asking: *"Generate test cases for the following code"*. The second unit is the Program Under Test (PUT) and the last unit is a set of instructions in a programming language named INS_2. The INS_2 acts as a hint to indicate the desired output for LLMC. The concatenation of (INS_1, PUT, INS_2) builds the initial prompt for *zero-shot* learning (Line 2 in Algorithm 2).

- *few-shot*: Prompt generation based on *few-shot* learning uses a chain of inputs and expected outputs related to the downstream task. There are different approaches for presenting the pair of input and output in the prompt. We follow a recent approach to build the initial prompt with *few-shot* strategy in *MuTAP* [39]. Considering the maximum possible length of tokens for LLMC (4k tokens in our study), *few-shot* prompt includes two different demonstrative examples of a Method (M), as an example PUT, and a Unit Test (UT) as follows (Line 5 in Algorithm 2):

```
<code>M_1</code>\n<test>UT_1</test>\n
<code>M_2</code>\n<test>UT_2</test>\n
<code>PUT_i</code>\n <test>
```

The methods provided in illustrative examples within the *few-shot* prompt serve as instances of PUT and UT containing their respective test cases. There is no natural language description of *PUT* in the initial prompt since such descriptions may not always be available, and *MuTAP* relies on the ability of LLMC to synthesize code context. *MuTAP* calls the initial prompt, *zero-shot* or *few-shot*, on LLMC and then passes the inferred output to the next step (Line 2 in Algorithm 1).

3.2. Refining

In this section, we describe the process of refining the generated test cases in *MuTAP* which includes fixing syntactical errors and intended behavior repair. The details are shown in Algorithm 3.

3.2.1. Syntax Fixer

The test cases generated by LLMC may have syntax errors (missing brackets, uncompleted lines, etc.). Listing 1 shows the unit test before the refining step generated by the LLMC for the PUT in the Motivating example from Fig. 1. The last line in this example has a syntax error. Since *MuTAP* needs to execute the test function for investigation on MT and prompt augmentation, samples with syntax errors become inefficient. However, sometimes a small change in the output of LLMC can fix the syntactic error and convert it into an executable test case for example completing the last line in Listing 1 or removing it.

```
def test():
    assert any_int(3, 2, 5) == True
    assert any_int(3, 2, 2) == True
    assert any_int(5.2, -2.2, 2) == True
    assert any_int(1, 2, 4 ==
```

Listing 1: The unit test before refining step for the PUT in the Motivating example presented in Fig. 1.

MuTAP starts this step by first compiling the test function, if any syntax error arises, *MuTAP* uses the capability of its LLMC to fix syntax errors, similar to other studies [35,40]. To do so, LLMC is called on a new prompt to fix the syntax error in its own output (Procedure *SyntaxFixer* in Algorithm 3). The syntax fixing prompt consists of two parts. The first part is a natural language instruction, INS_{fix}, *"Fix the syntax errors in the following code snippet"*, and the second part is the generated test function by LLMC on the initial prompt (Line 7–8 in Algorithm 3). If the syntax error persists even after re-prompting the LLMC, *MuTAP* employs the Python parser to identify the erroneous line. It then retains the lines preceding the problematic line, ensuring they remain free of syntax errors (Line 13 in Algorithm 3).

3.2.2. Intended behavior repair

Based on the initial prompt, LLMC generates different test cases that are serialized as an assertion oracle by calling the *PUT* on certain inputs and comparing the returned output of *PUT* with the expected output or ground truth, for example, {`assert any_int (3, 2, 5) == True`}. However, it is possible for the LLMC to generate test cases that are asserting wrong test output. It means that for some test cases, LLMC does not generate the expected return output of the *PUT*. The lack of a natural language description about the *PUT* in the initial prompt could potentially lead to the generation of test cases that do not accurately reflect the intended behavior of the method. In Listing 1, the initial test case asserts a correct test output, whereas, in the second and third test cases, the test output is incorrect.

The assertion with wrong test output may fail on mutants, not because of detecting the bug, but because of the unintended behavior of the assertion. These failures cause confusion about the effectiveness of test cases. So, this step of *MuTAP* aims at repairing the intended behavior of assertion oracles in the test cases (Procedure *IntendedBehaviorFixer* in Algorithm 3).

For each assertion in the test, *MuTAP* runs the *PUT* over the test inputs and compares the return output of *PUT* with the asserting output. If the returned output of *PUT* is the same as the asserting output in the oracle, then *MuTAP* considers it as an assertion oracle with the correct intended behavior. Otherwise, it repairs those assertions by replacing the asserting output with the expected output of *PUT* (Line

Algorithm 3: Refining

```
   Input: raw_IUT, PUT
   Output: IUT // The Initial Unit Test after refining
1  syntax_fixed_IUT ← SyntaxFixer (raw_IUT)
2  IUT ← IntendedBehaviorFixer (syntax_fixed_IUT, PUT)
3  return IUT
4
5  Procedure SyntaxFixer(raw_IUT)
6      if not AST.parse(raw_IUT) then
7          syntax_fixed_prompt ← CONCAT (INS_fix, raw_IUT)
8          syntax_fixed_IUT ← LLMC (syntax_fixed_prompt)
9      end
10     syntax_fixed_IUT ← SyntaxCheck (syntax_fixed_IUT)
11     return syntax_fixed_IUT
12
13 Procedure SyntaxCheck(syntax_fixed_IUT)
14     if AST.parse(syntax_fixed_IUT) then
15         return syntax_fixed_IUT
16     else
17         return SyntaxCheck (syntax_fixed_IUT [:error_line])
18     end
19
20 Procedure IntendedBehaviorFixer(syntax_fixed_IUT, PUT)
21     fixed_IUT ← { }
22     foreach test_case ∈ syntax_fixed_IUT do
23         expected_output ← PUT(test_case.input)
24         if expected_output ≠ test_case.output then
25             test_case.output ← expected_output
26         fixed_IUT.append(test_case)
27     end
28     return fixed_IUT
```

22–27 in Algorithm 3). *MuTAP* omits those assertions for which the input types failed on *PUT*, for example, if *PUT* expected a `list` of integers but the test input is a `string`. The final outcome of this step is named *Initial Unit Test (IUT)* which is a set of test cases generated by LLMC after refinement as shown by 2 in Fig. 1 for the unit test in Listing 1.

3.3. Mutation Testing (MT)

MT assesses the quality and effectiveness of test cases. Mutants are built by injecting artificial bugs into the *PUT* to simulate defects. If test cases failed on a mutant, we consider it as a killed mutant, otherwise, it survived, meaning that the test cases within the unit test are not able to detect it. The presence of surviving mutants highlights the shortcomings of test cases, suggesting the need to either add a new test case or improve an existing one. The *Mutation Score (MS)* represents the effectiveness of test cases by calculating the ratio of killed mutants out of all mutants of a *PUT*.

Algorithm 4 presents the details of this step. Similar to other studies [41], *MuTAP* uses MutPy [42] to generate different mutants for each *PUT* and calculate *MS* (Line 3–7 in Algorithm 4). Executing test cases on each mutant involves performing some preliminary setups. For this purpose, *MuTAP* uses Python's built-in "setuptools.find_packages" to locate and install the required packages, such as "math", "numPy", "pandas", "pytest", and others. Additionally, *MuTAP* implements setup functions that are responsible for creating temporary directories, which are utilized during the execution of the test cases on the mutants. After executing the test cases on the mutants and calculating the *MS*, *MuTAP* properly tears down the setup by removing the temporary directory.

As shown on Line 5–9 in Algorithm 1, if the *MS* of a *PUT* reaches *100%*, *MuTAP* passes test cases to the oracle minimization step (Section 3.5 and Line 16 in Algorithm 1), otherwise, it collects the list of surviving mutants and transfers them to the prompt augmentation step (Section 3.4).

3.4. Prompt augmentation

Algorithm 5 shows the details of this step. If there is any surviving mutant from the previous step, *MuTAP* augments the initial prompt,

Algorithm 4: MutationTesting

```
  Input: UT, mutants
  Output: MS, surviving_mutant
1 surviving_mutant ← { }
2 foreach mut ∈ mutants do
3     if exec(mut, UT) then
4         surviving_mutant.append(mut)
5     end
6 end
7 MS ← (#(mutants) − #(surviving_mutant))/#(mutants)
8 return MS, surviving_mutant
```

Algorithm 5: AugmentingPrompt

```
  Input: initial_prompt, UT, SM
  Output: augmented_prompt
1 augmented_prompt ← CONCAT (initial_prompt, UT, INS_3, SM, INS_4)
2 return augmented_prompt
```

zero-shot or *few-shot*, by adding four new components (Line 1 in Algorithm 5). The first component is the *IUT*, the initial unit test generated by LLMC after refinement. The second component is an instruction in a natural language named INS_3 that clarifies the shortcoming of *IUT* by *"The test function, test(), cannot detect the fault in the following code"*. The third component is one of the surviving mutants of the *PUT*, named *SM*. The last component, INS_4 is an instruction in natural and programming language: the natural language context clarifies the task by asking to *"Provide a new test case to detect the fault in prior code"* and the programming language context acts only as a hint to guide LLMC for generating the output. An example is shown by 3 in Fig. 1.

MuTAP re-prompt LLMC and repeats the refining step on the generated output. Then, it appends new generated test cases to the *IUT* that we call augmented unit test (Line 10–12 in Algorithm 1). The augmented unit test and the mutants of *PUT* is passed to the *MT* step (Line 13 in Algorithm 1). *MuTAP* recursively repeats prompt augmentation till either the final test cases kill all the mutants ($MS = 100\%$) or there is no surviving mutant that is not used in the augmentation process (Line 14 in Algorithm 1). An example of updating the mutant component in Fig. 1, 3 is changed to 3′ by replacing SM_0 with SM_1. The 4′ indicates the generated test cases with LLMC after iterating the process on the next surviving mutant.

3.5. Oracle Minimization

The test cases generated by the LLMC usually consists of redundant assertions. Also, the augmentation process may add more redundant assertions to the final unit test. Presenting all of them (with redundancy) as the final output can cause confusion for developers. In the final step, similar to previous tools that generate mutation-driven test oracles [9,43], *MuTAP* minimizes the number of assertions by utilizing a Greedy technique to eliminate the redundant assertions that do not improve the MS. This step is presented in Algorithm 6. *MuTAP* starts by tracking the number of mutants that each assertion kills and then chooses the test case containing the assertion that kills the maximum number of mutants. This process is then repeated by adding the test cases containing the next assertions that detect the most mutants (Line 4–10 in Algorithm 6). If adding this new assertion increases the MS, *MuTAP* keep the test case and its assertion. Otherwise, the test case will be discarded as redundant.

4. Evaluation

In this section, we describe the evaluations we designed and conducted to investigate the following research questions:

Algorithm 6: OracleMinimization

```
Input: UT, mutants
Output: FUT
MS_old ← 0
FUT ← {}
sorted_UT ← sort(UT) // sort test cases in UT based on the
    descending order of MS
foreach test_case ∈ sorted_UT do
    FUT.append(test_case)
    MS, surviving_mutant ← MutationTesting (FUT, mutants)
    if MS > MS_old then
        MS_old ← MS
    else
        FUT.delete(test_case)
    end
end
return FUT
```

RQ1 How effective are test cases generated by *MuTAP* in comparison to test cases generated by automatic test generation tools?
In this RQ, we aim to assess test cases in revealing bugs. We compare *MuTAP*-generated test cases with a state-of-the-art automatic test generation tool and also the output of LLMC before refining and augmenting steps as baselines. We evaluate the effectiveness of test cases on synthetic and real buggy code.

RQ2 How do the different parts of *MuTAP* perform?
In this RQ, we aim to assess the individual impact of different components in our proposed method, *MuTAP*, on the improvement of syntax fixing, repair intended behavior, and test case effectiveness.

RQ3 What is the performance of *MuTAP* for each mutation type?
Since we utilize various types of mutants in the prompt augmentation step, it is worth exploring whether *MuTAP* performs differently for each type. Through this RQ, we aim to ascertain if the improvement is limited to a specific mutant type or if it is broadly observed across various mutant types.

4.1. Experimental setup

In this section, we present our experiment setup. We describe the automatic test generation tool used to compare our results, clarify the LLMC of *MuTAP* and its setup, and indicate the baselines and datasets used in our experiments.

We conducted the experiment on the Cedar cluster of Compute Canada, which offers 32 cores CPU, 1TB storage, and one v100l GPU with 32 GB GPU Memory, and on a system running Linux 5.15.0-69-generic with AMD FX(tm)-6300 Six-Cores CPU, 512 GB storage, and 16 GB Memory.

4.1.1. Experimental settings

We conducted an experiment for each PUT. In each experiment, we repeated the runs 10 times for *MuTAP* with different configurations (i.e., Codex and zero-shot) and also Pynguin. The MS reported is the median of the 10 runs. When selecting unit test candidates from the output generated by LLMC in different components of *MuTAP*, we considered two criteria: the candidate should contain both the keywords *assert* and the *function name* of the *PUT*. If, after 10 runs, LLMC failed to generate an output containing these two keywords, we categorized the PUT as problematic or as a case for which *MuTAP* cannot generate a test case. We followed the same approach for Pynguin; if, after 10 runs, Pynguin does not generate a test case for a PUT, we categorize the PUT as problematic.

We also have an internal repetition in each run, regarding the syntax fixing step. We run the syntax fixing prompt for each unit test on the LLMC for up to 10 iterations. If the syntax error remains unresolved even after 10 iterations, *MuTAP* employs the Python parser to locate the erroneous line. It then retains the lines preceding the buggy line, ensuring their freedom from syntax errors. If the removal of lines results in the absence of any remaining test cases (all test cases prove non-compilable), we classify the PUT as problematic.

We also conducted a statistical test to determine whether the effectiveness of test cases generated by *MuTAP* is significantly different from those generated by Pynguin and other comparable methods. Due to the importance of killed/detected mutants/buggy code snippets in evaluating the effectiveness of test cases and due to the limited number of PUTs in our datasets, we conducted the statistical test on categorical data to determine if the proportion of mutant/buggy code snippets that were killed/detected by *MuTAP* is significantly different from other comparable methods. To apply the statistical test on the categorical data, we employed the Chi-square test with a df of 1 and a p-value threshold of 5% (0.05). The result of the statistical test indicates if the proportion of mutant/buggy code detected by *MuTAP* is significantly different from other comparable methods. In this statistical test, the null hypothesis states that there is no significant difference between the proportion of mutant/buggy code detected by *MuTAP* compared to other methods (p-value > 0.05). A p-value < 0.05 indicates that the null hypothesis can be rejected and the improvement in the effectiveness of test cases generated by *MuTAP* is statistically significant. We also calculated the effect size on the proportion of detected mutant/buggy code (based on the chi-square test) and reported the magnitude of the effect size on the proportion of detected mutant/buggy code snippets [44,45].

4.1.2. Comparable tool

Pynguin [41] is a well-known fully-automated test generation tool for a dynamically typed programming language such as Python. It uses different search-based algorithms to satisfy code coverage criteria, i.e., branch coverage. Pynguin first takes a Python code (method, module, etc.) as input and collects its information such as variable types, method names, and dependencies. Then it uses one of the search-based test generation algorithms (MIO [46], MOSA [47], DynaMOSA [48], etc.) to generate test cases. It randomly mutates (deletes, inserts, replaces) different values and statements within the test case to generate new test cases and executes them over the *PUT* to ensure their correctness. Finally, it generates assertions for test cases using a MT engine [41].

For our experiments, we employ Pynguin 0.17.0. with the DynaMOSA [48]. According to the evaluation of Pynguin [48], DynaMOSA shows the best performance compared to the other algorithm in generating test cases with this tool. We set the timeout of test generation to 600 s which is the default setting of the tool.

4.1.3. Large Language Model Component (LLMC)

We employ two different LLMs as the LLMC of *MuTAP*. The first one is OpenAI's Codex, designed specifically for code generation tasks [17]. We use *Code-davinci-002*, with a temperature of 0.8. The lower temperature causes less variation in the outputs of the model while the higher temperature increases the variation of output and then the chance of generating useful test cases over different iterations. The evaluation of CODAMOSA [16] shows that 0.8 is a reasonable temperature to generate useful test cases with Codex.

The second LLM is Meta's *llama-2-chat*, which has been iteratively refined using Reinforcement Learning with Human Feedback (RLHF) and is appropriate for dialog use cases [49]. Similar to Codex, we have configured the model's temperature to be 0.8. Furthermore, the model provides three distinct roles as the structure of the prompt: *system, user,* and *assistant*. These roles serve the purpose of clarifying each component of the prompt to the model by assigning specific components to each role. This structure follows the model training procedure. The *System* role defines the model's behavior, designating it as an assistant with a specific type of task. For instance, it clarifies whether the model operates as a Python programming assistant or serves as a poet that tries to find memorable names for variables. Given the model's training for conversational (chat) setup, the *user* role is assigned to the user's

Table 1
List of the mutation operators in our experiments used by $MutPy$ sorted by alphabetical order.

Operator	Example	Mutant
AOD — arithmetic operator deletion	result.append(numbers[−1])	result.append(numbers [1])
AOR — arithmetic operator replacement	return number % 1.0	return number * 1.0
ASR — assignment operator replacement	current_depth + = 1	current_depth − = 1
BCR — break continue replacement	if i % j != 0: break	if i % j != 0: continue
COD — conditional operator deletion	if not string: return ' '	if string: return ' '
COI — conditional operator insertion	if balance < 0: return True	if (not balance < 0): return True
EHD — exception handler deletion	except: pass	except: raise
EXS — exception swallowing	except: return False	except: pass
LCR — logical connector replacement	if s[−1] == 'y' or s[−1] == 'Y':	if s[−1] == 'y' and s[−1] == 'Y':
ROR — relational operator replacement	if c[n] ≤ 1:	if c[n] ≥ 1:
SIR — slice index remove	l[::3] = sorted(l[::3])	l[::3] = sorted(l[:])

prompt or request, while *assistant* encompasses the model's response related to the user's prompt. Different combinations of these roles can be utilized in each prompt to tailor the interaction with the model according to the specific requirements [49].

In our experiments, the role of the *system* is defined as *{You are a Python coding assistant. Always answer with Python code.}*, for all types of prompts, including *zero-shot*, *few-shot*, and *augmented* prompts. To handle the *zero-shot* prompt, we only set the *user's* role content to be a concatenation of *(*INS_1*,* PUT_i*,* INS_2*)*. For the *few-shot* prompt, we define the content of the *assistant* role as a set of demonstrative examples of Method (M) and Unit Test (UT), while the *user* role content is set to PUT_i. As for the *augmented* prompt, its various components are set up as follows:

```
{user: Initial Prompt,
assistant: IUT,
user: concat(INS3, SMi, INS4)}
```

For both LLMs, the maximum number of generated tokens is set to 250 for generating test cases and 20 tokens for syntax fixing, based on previous studies on similar tasks [16,50]. The stop word is defined as *quote (")* for *zero-shot* and as $</test>$ for *few-shot* prompt. For the rest of the hyperparameters, we keep the model's default values.

It is important to note that *MuTAP* is not limited to these two models, and its LLMC can be replaced with any other LLM as required.

4.1.4. Baselines

In addition to Pynguin, we propose two baselines for each LLM to evaluate our proposed method, *MuTAP*.

Before-refining: The first baseline is the output of the initial prompt on LLMC (Codex or llama-2-chat), without fixing syntax errors or repairing the intended behavior. Since assertions with unintended return values can fail on mutants or buggy code and present invalid effectiveness, we omit those assertions in this baseline to avoid this side effect. If the output of the model has syntax errors, we consider it as a wrong test and consequently consider the task as a problematic or unsolved task.

After-refining: The second baseline is the output of the initial prompt on LLMC (Codex or llama-2-chat), after applying the following steps: *Refining* (Section 3.2) and *Oracle Minimization* (Section 3.5).

4.1.5. Mutant generator

To apply MT, we need to generate different mutant versions of a *PUT* by injecting bugs into its different lines. For this purpose, we use *MutPy* version 2.0 [42]. *MutPy* is a MT tool for code in Python 3.3+. It benefits from different mutation operators to generate the mutants. The list of mutation operators used in our experiment with corresponding examples is shown in Table 1. *MutPy* injects one operator at a time to generate the mutant if the operator is applicable on *PUT*.

Table 2
Datasets used for evaluation.

Dataset	# PUTs	Mutants\Bugs	Description
HumanEval [17]	164	1260	Synthetically generated bugs from PUTs
Refactory [31]	5	1710	Real student buggy code on 5 assignments

4.1.6. Datasets

To conduct our experiments, we use two different datasets as shown in Table 2. The first one is *HumanEval* [17] which is a dataset to evaluate LLMs that generate code. It has 164 human-written programming problems at easy to medium levels. We consider each programming problem in this dataset as a PUT. Each problem has different attributes such as descriptions and reference solutions. We use the reference solution of each task as a *PUT*. The PUTs in the HumanEval dataset are either self-contained or dependent on public libraries, such as NumPy. The average number of lines of code (LOC) across all the PUTs in this dataset is 11.5.

The second one, *Refactory* [31], is a benchmark for Python bug repairing [51]. It has 1710 buggy students' submissions for 5 assignments of a Python programming course. Each assignment has a correct reference solution that we use as *PUT*. The PUTs in the Refactory dataset are all self-contained. The average number of LOC across all the PUTs in this dataset is 8.4. The advantage of this dataset is buggy code snippets generated by humans that give us the opportunity to evaluate test cases generated by *MuTAP* on real bugs and compare them with Pynguin and our baselines.

Both datasets employed in this study are in Python. Python is a very common and widely used programming language across various domains [52], making it a representative and well-suited choice for our investigation. Moreover, various LLMs show better performance in synthesizing Python code. Notably, LLMs like Codex are specifically trained and optimized for this language. *MuTAP* can be leveraged for Java as it is language-agnostic by design.

4.2. Experimental results

In this section, we discuss our findings for each RQ.

4.2.1. RQ1: How effective are test cases generated by MuTAP in comparison to test cases generated by automatic test generation tools?

Since our study focuses on MT to improve the effectiveness of test cases, we compare *MuTAP* with Pynguin and our baselines in terms of MS, number of killed mutants, and number of *PUT* with 100% MS. The MS reflects the effectiveness of test cases generated by the different methods in revealing bugs, excluding PUTs for which the method failed to generate test cases (problematic PUTs). However, the number of killed mutants, in comparison to the total number of mutants (1260), provides insight by reflecting the total number of surviving mutants for both PUTs with test cases and those without. Given that the goal of

Table 3
Evaluation result of test cases generated by *MuTAP* and other methods on *synthetic* buggy programs on the HumanEval. *"MS (Avg.)"* represents the average of MS with its standard deviation over all PUTs in the HumanEval dataset. The MS for each PUT is the median of 10 runs. *"Problematic PUT"* refers to the percentage of PUTs without an accurate test case and *"PUT with MS = 100%"* denotes the percentage of PUTs for which their test cases achieve an MS of 100%, for both metrics out of 164 PUTs. *"Killed Mut"* also represents the absolute number of killed mutants for all PUTs.

Prompt	Model	Method	# test cases (Avg.)	Problematic PUT (%)	MS (Avg.) (%) ± *std*	# Killed Mut (out of 1260)	Effect Size (ES)	PUT MS = 100% (%)
–	–	Pynguin	1.5 (min = 1, max = 4)	18.9	65.94 ± 30.78	649***	*Large*	28.22
Zero-shot	Codex	Before-refining	1.5 (min = 1, max = 3)	44.51	72.15 ± 26.95	296***	*Large*	11.04
		After-refining	2.1 (min = 1, max = 3)	18.29	76.82 ± 24.35	749**	*Medium*	24.54
		MuTAP	**2.5 (min = 1, max = 4)**	**18.29**	$89.13\% \pm 20.32$	**869**	–	**41.72**
Zero-shot	llama2-chat	Before-refining	1.2 (min = 1, max = 3)	41.46	$62.60\% \pm 28.82$	318***	*Large*	17.79
		After-refining	2.2 (min = 1, max = 5)	0	$84.04\% \pm 17.41$	1059*	*Small*	53.98
		MuTAP	**2.5 (min = 1, max = 5)**	**0**	$91.98\% \pm 13.03$	**1159**	–	**68.09**
Few-shot	Codex	Before-refining	1.5 (min = 1, max = 3)	23.78	$72.68\% \pm 26.23$	508***	*Large*	15.95
		After-refining	2.2 (min = 1, max = 3)	16.46	$82.73\% \pm 21.91$	829**	*Medium*	34.97
		MuTAP	**2.6 (min = 1, max = 7)**	**16.46**	$92.02\% \pm 13.55$	**922**	–	**49.69**
Few-shot	llama2-chat	Before-refining	1.5 (min = 1, max = 3)	36.58	$64.51\% \pm 24.11$	325***	*Large*	22.69
		After-refining	2.5 (min = 1, max = 5)	0	$85.16\% \pm 16.36$	1073*	*Small*	57.05
		MuTAP	**2.6 (min = 1, max = 7)**	**0**	$93.57\% \pm 11.18$	**1179**	–	**69.93**

The result of the Chi-square test with a *p*-value threshold of 5%. Significant at the *** $p < 0.001$, ** $p < 0.01$, *$p < 0.05$.
The magnitude of the effect size at $ES < 0.1$ negligible, *Small* $ES \geq 0.1$, *Medium* $ES \geq 0.3$, *Large* $ES \geq 0.5$.

MT is to generate test cases with MS=100%, the total number of PUTs with a 100% MS demonstrates the improvement achieved by MuTAP in this aspect compared to Pynguin and the baselines. Absolute values in Tables 3 and 4 are reported based on the unit tests with the median MS.

HumanEval dataset

Table 3 shows the obtained results for the *HumanEval* dataset. Prior to syntax fixing and intended behavior repair (*before-refining*), the test cases generated by Codex and llama-2-chat are incorrect for 44.51% and 41.46% of *PUT*s, respectively, when using the *zero-shot* initial prompt. However, they managed to kill 295 and 318 mutants (out of 1260), respectively.

The initial prompt has a more pronounced impact on the output of Codex compared to llama-2-chat. Switching the initial prompt to *few-shot* decreases the number of *PUT*s without test cases to 23.78%, while also raising the number of killed mutants to 508 when using Codex as LLMC. On the other hand, when using llama-2-chat, the number of *PUT*s without test cases reduces to 36.58%, and the number of killed mutants increases from 318 to 325. This difference in performance could be attributed to llama-2-chat being more suitable for dialog prompts, and using a prompt with a pair of demonstrative input and output, devoid of natural language context, does not improve the model's performance significantly.

In contrast, Pynguin, as the state-of-the-art automatic test generation tool, outperforms the output of both LLMs, before-refining, by killing 649 mutants and failing to generate test cases for 18.9% *PUT*s.

After applying the post-processing steps of syntax fixing and intended behavior repair, *MuTAP* with both LLMs perform better than Pynguin in terms of killing more mutants. Notably, when using both *zero-shot* and *few-shot* prompts, llama-2-chat is able to generate correct test cases for all *PUT*s, after-refining. However, their effectiveness in terms of killing mutants is measured at 84.04% and 85.16% with the *zero-shot* and *few-shot* prompts, respectively.

On the other hand, the MS of test cases generated by Codex after refining is 76.82% and 82.73% with the *zero-shot* and *few-shot* prompts, respectively. Despite this improvement, Codex still fails in generating correct test cases for 18.29% (with *zero-shot*) and 16.46% (with *few-shot*) of *PUT*s after refining.

MuTAP, enhances the effectiveness of test cases generated by both LLMs, Codex, and llama-2-chat, achieving MS of 89.13% and 91.98% with the *zero-shot* prompt, and an MS of 92.02% and 93.57% with the *few-shot* prompt, respectively. Particularly, *MuTAP* with the *few-shot* prompt when using llama-2-chat as its LLMC manages to kill 1179 mutants out of 1260 and generates test cases with MS=100% for up to 70% of *PUT*s, demonstrating a remarkable improvement in the effectiveness of test cases compared to the Pynguin with 649 killed mutants and 28.22% *PUT*s with MS=100%. As the results of our statistical test on the portion of killed mutants over all PUTs also show, the effectiveness of test cases generated by *MuTAP* is significantly different from those generated by other comparable methods (all *p*-values are below 0.05). Also, the magnitude of the effect size on the proportion of mutants killed by MuTAP with different configurations, compared to other alternative methods, is non-negligible.

Refactory dataset

To evaluate the performance of *MuTAP* on buggy programs, we employ the *Refactory* dataset. To evaluate the results on this dataset, we select the unit tests with a median MS out of 10 runs generated by different methods and apply them to the buggy code in *Refactory* to assess their effectiveness in detecting real buggy code snippets. We report the absolute total number of buggy code snippets over all PUTs that are detected by unit tests with the median of MS generated by different methods.

Table 4 that shows the results on this dataset confirms our findings on *HumanEval*. Overall, our proposed method, *MuTAP*, detects more buggy programs compared to Pynguin and other baseline methods.

MuTAP with *few-shot* learning while using llama-2-chat identifies 468 more buggy code compared to Pynguin and 111 more buggy code compared to *After-refining*. Furthermore, *MuTAP* while using llama-2-chat as its LLMC discovers 79 buggy code that was not detected by either Pynguin or llama-2-chat's test cases *After-refining* process. When using Codex, *MuTAP* detects 73 buggy code that was missed by both Pynguin and Codex's test cases *After-refining* stage. Moreover, *MuTAP* excels in generating more effective test cases, with an average of 2.6 test cases after applying greedy optimization. As the results of our statistical test on this dataset also show the proportion of buggy code detected by *MuTAP* is significantly different from other comparable methods. The stars indicate the degree of statistical significance over the alternative methods (always compared to *MuTAP* with a specific configuration). Moreover, the magnitude of the effect size on the proportion of buggy code detected by *MuTAP* with different configurations, compared to other alternative methods, is non-negligible.

```
def derivative(xs: list):
    return [(i * x) for i, x in enumerate(xs)
    ][1:]
```

Listing 2: A sample PUT for which MuTAP, incorporating Codex, was unable to generate test cases.

```
def test_case_0():
    float_0 = 890.6
    list_0 = [float_0, float_0, float_0, float_0]
    var_0 = derivative(list_0)
    assert len(var_0) == 3
```

Listing 3: The only test case generated by Pynguin for the PUT in Listing 2.

```
def test():
    assert derivative([1]) == []
    assert derivative([1, 2, 3]) == [2, 6]
    assert derivative([1, 2, 3, 4]) == [2, 6, 12]
    assert derivative([3, 1, 2, 4, 5]) == [1, 4, 12, 20]
    assert derivative([1.0, 2.0, 3.0] == [2.0, 6.0]
    assert derivative(['a', 'b', 'c', 'd', 'e']) == ['b', 'cc', 'ddd', 'eeee']
```

Listing 4: The test cases generated by MuTAP, incorporating llama-2-chat, for the PUT in Listing 2.

The challenges that Pynguin faces in generating valid and effective test cases for certain PUTs can be attributed to some of its limitations. While Pynguin is capable of generating test cases for self-contained PUTs and PUTs with dependencies on public libraries, such as NumPy, it sometimes exhibits limitations in generating effective tests for such PUTs. However, this limitation is not observed when LLMs are employed in *MuTAP* for test case generation. Moreover, for PUTs that incorporate generators like ‘yield’ or iterators such as list comprehensions, Pynguin encounters difficulties in generating corresponding test cases. In contrast, *MuTAP* demonstrates no limitations in generating test cases for PUTs with such constructs.

In addition, type information is crucial for generating high-quality test cases. While this information is not available in dynamically typed languages like Python, Pynguin extracts some of the type information during its test case generation process if available. However, one of the factors impacting Pynguin's ability to generate effective test cases is the potential presence of incorrect, incomplete, or lacking type information. In *MuTAP*, the semantics of function and variable names (input/output names) in PUTs enable LLMC to make assumptions about variable types that help in generating more effective test cases which Pynguin could not benefit from.

Conversely, *MuTAP* encounters challenges in generating effective test cases for PUTs where syntax does not provide comprehensive information about their functionalities, as no description of the functionality is incorporated in the prompt. An illustrative example of such PUTs is demonstrated in Listing 2, where *MuTAP*, incorporating Codex, is unable to generate test cases for this PUT. In contrast, Listing 3 shows the only test case generated by Pynguin for the same PUT in Listing 2. Although the PUT involves an iterator within a list comprehension, Pynguin can generate test inputs and capture the test output. However, the assertion compares the length of the output list with an expected value, which does not accurately represent the functionality of the PUT in Listing 2. The MS for this test case generated by Pynguin is 25%. Listing 4 presents the unit test generated by *MuTAP*, incorporating llama-2-chat, for example PUT in Listing 2. The test cases within this unit test are more comprehensive in evaluating the functionality of the PUT and prove more effective in revealing bugs, achieving an MS of 100%.

Overall, *MuTAP* using both llama-2-chat and Codex demonstrates better performance compared to Pynguin in terms of killing mutants and detecting buggy code. The effectiveness of these test cases in detecting bugs is improved through post-processing steps of refining and prompt augmentation.

> **Finding 1:** *MuTAP* generates more effective test cases compared to Pynguin and conventional *zero-shot* and *few-shot* learning on LLM. The number of *MuTAP*'s test cases is not much greater than the output of other methods after minimization. Additionally, LLM with dialog setup performs better on the augmented prompt. In conclusion, the effectiveness of LLM-generated test cases can be enhanced through prompt augmentation using surviving mutants and post-processing refinement.

4.2.2. RQ2: How do the different parts of *MuTAP* perform?

Syntax Fixer: On average, the percentage of test cases with syntax errors is 38.98% and 26.48% when using the *zero-shot* and *few-shot* prompts, respectively, with Codex. When employing llama-2-chat, this percentage is 33.85% and 26.32% with the *zero-shot* and *few-shot* prompts, respectively.

When considering syntax errors, three factors contribute to decreasing them in the output of LLMs. The first factor is the type of initial prompt. As shown in Table 5 on the *HumanEval* dataset, *few-shot* learning results in fewer syntax errors in the output of both LLMs. Specifically, when using Codex, the percentage of syntax errors decreases from 44.79% to 29.03% after-refining, and for *MuTAP*, it decreases from 33.17% to 23.93%. With llama-2-chat as the LLMC, the percentage of syntax errors decreases from 38.03% to 26.99% after refining, and from 29.66% to 25.64% for *MuTAP*.

The second impactful factor, which is also the primary factor, is the *Syntax Fixing* component. As shown in Table 5, when using Codex, this component in *MuTAP* on average fixes 14.5% of syntax errors by utilizing the LLMC and addresses 81.37% of syntax errors by omitting the lines causing the errors. On the other hand, when using llama-2-chat as the LLMC of *MuTAP*, the *Syntax Fixing* component, on average, resolves 32.31% of syntax errors through re-prompting the LLMC, and 60.73% of the errors by omitting the problematic lines.

The final factor contributing to the improvement of syntax errors in test cases is the prompt augmentation process in *MuTAP*. By augmenting the prompt with *IUT*, the occurrence of syntax errors in the output of Codex with the *zero-shot* technique decreases from 44.79% to 33.17%. Similarly, with llama-2-chat and the *zero-shot* prompt, the percentage of syntax errors reduces from 38.03% to 29.66%. Augmenting the prompt with *IUT* provides illustrative examples of test cases and serves a similar purpose to the demonstrative examples in the *few-shot* learning prompt, effectively reducing syntax errors in the output of LLMs.

Our finding on the *Refactory* dataset shows *MuTAP* generates test cases with syntax errors in only one *PUT* (out of 5) using Codex and *zero-shot* learning. Moreover, none of those syntax errors could be fixed by re-prompting LLMC. On the other hand, for both initial prompt types, syntax errors decrease to zero using llama-2-chat.

Intended Behavior Repair: In the case of repairing intended behavior, two distinct factors contribute to reducing the error rate in assertion oracles. As shown in Table 6, the *Intended Behavior Repair* step, when using Codex as the LLMC, on average, fixes 83.98% (82.21% with *zero-shot* and 85.75% with *Few-shot*) and 89.86% (89.71% with *zero-shot* and 90.00% with *Few-shot*) of incorrect behaviors in the *after-refining* and *MuTAP*, respectively. When utilizing llama-2-chat, this step repairs, on average, 84.35% and 95.96% of unintended behavior in the *after-refining* and *MuTAP*, respectively.

Table 4
Evaluation results on buggy programs on the *Refactory* dataset. The *"Bug Detected"* column shows the absolute number of *real* buggy programs detected by each Method. *"MS (Avg.)"* represents the average of MS with its standard deviation over 5 PUTs in the Refactory dataset. The MS for each PUT is the median of 10 runs.

Prompt	Model	Method	# test cases (avg)	MS (Avg.) (%) $\pm$ *std*	Bug Detected (out of 1710)	Effect Size (ES)
–	–	Pynguin	1.25 (min = 1, max = 4)	55.93 ± 32.45	1155***	*Large*
Zero-shot	Codex	After-refining	1.2 (min = 1, max = 2)	66.11 ± 17.77	1356**	*Medium*
		MuTAP	**1.6 (min = 1, max = 3)**	77.91 ± 19.24	**1437**	–
Zero-shot	llama-2-chat	After-refining	1.2 (min = 1, max = 3)	76.93 ± 12.97	1478***	*Medium*
		MuTAP	**2.2 (min = 1, max = 4)**	94.40 ± 11.20	**1594**	
Few-shot	Codex	After-refining	1.6 (min = 1, max = 3)	67.93 ± 16.94	1411**	*Medium*
		MuTAP	**2.2 (min = 1, max = 4)**	83.73 ± 14.31	**1529**	–
Few-shot	llama-2-chat	After-refining	2.1 (min = 1, max = 4)	76.93 ± 12.97	1512***	*Medium*
		MuTAP	**2.2 (min = 1, max = 4)**	94.40 ± 11.20	**1623**	–

The result of the Chi-square test with a *p*-value threshold of 5%. Significant at the *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$.
The magnitude of the effect size at $ES < 0.1$ negligible, *Small* $ES \geq 0.1$, *Medium* $ES \geq 0.3$, *Large* $ES \geq 0.5$.

Table 5
Syntax error fixing of test cases. The syntax error rate shows the ratio of unit tests with syntax errors.

Model	Method	Prompt	# iteration (avg)	Syntax error rate	Fixed by model	Fixed by omitting lines
Codex	After-refining	Zero-shot	9.1	44.79%	16.44%	60.27%
		Few-shot	9.5	29.03%	12.96%	83.33%
	MuTAP	Zero-shot	9.7	33.17%	16.18%	79.41%
		Few-shot	9.5	23.93%	12.82%	84.62%
llama2-chat	After-refining	Zero-shot	7.1	38.03%	30.64%	63.86%
		Few-shot	6.8	26.99%	31.81%	57.96%
	MuTAP	Zero-shot	6.9	29.66%	32.17%	61.05%
		Few-shot	6.8	25.64%	32.45%	60.40%

In addition to the *Intended Behavior Repair* step, the prompt augmentation step in *MuTAP* significantly reduces the occurrence of unintended behavior in test cases. In Table 6, it is shown that when for example llama-2-chat is employed as the LLMC alongside *Few-shot* initial prompt, 63.25% of test cases generated by the model have assertion errors. This indicates that within this context, about 36.75% of the test oracles generated by the model demonstrate the correct behavior, including accurate test outputs. Subsequently, this figure drops to 10.75% after augmenting the initial prompt with the *IUT* and surviving mutants. This demonstrates that 89.25% of the test oracles generated by *MuTAP* have the correct behavior, encompassing accurate test outputs.

When using Codex with a *zero-shot* prompt, the assertions with unintended behavior, such as wrong test output, decrease from 63.63% to 19.38%. Similarly, with llama-2-chat and using *few-shot* prompt, the assertions with unintended behavior decrease from 63.25% to 10.75%. The reason behind this improvement could be attributed to the usage of *IUT*s (Initial Unit Tests) in *MuTAP* for augmenting the initial prompt. These *IUT*s already represent the intended behavior of the PUT, thereby assisting the LLM in suggesting test cases with less unintended behavior (i.e., fewer wrong test outputs). Also, on the *Refactory* dataset, *MuTAP* repaired all assertions with incorrect behavior on the output of both initial and augmented prompts.

Unlike syntax errors, the prompt type does not significantly help with unintended behavior in assertions. The combination of the *Intended Behavior Repair* step and the prompt augmentation process improves the effectiveness of test cases, ensuring that they align with the intended behavior of *PUT*.

Surviving Mutants Representation: We also investigated the impact of surviving mutants' order on MS during prompt augmentation. Fig. 3 illustrates the effect of augmenting the prompt with a random order of surviving mutants over 5 runs for all *PUT*s. For this comparison, we randomly selected one of the surviving mutants of each *PUT* with $MS < 100\%$ and utilized it to augment the initial prompt. We then calculated the average MS for all *PUT*s. Subsequently, we randomly chose the second surviving mutant for the remaining *PUT*s with MS < 100% (if any), repeated the augmentation process as a second iteration, and calculated the average MS for all *PUT*s again. We continue to iterate this process (axis in Fig. 3) until either there are no more *PUT*s with MS < 100% or no more surviving mutant that is not utilized in the argumentation process.

As shown in Fig. 3, each data point represents an iteration of the augmentation step and the average MS for all *PUT*s across five runs, derived from a random selection of surviving mutants. The shaded area illustrates the standard error of the average MS across these five runs in each iteration. The results show that the standard error over 5 runs in each iteration is not significant. However, during the initial iterations (up to 7 iterations), the standard error around the average MS over five runs is greater than what is observed in the final iterations. That is because after several iterations over the augmentation step with different surviving mutants, the improvement in MS stalls. Notably, more than 90% of the MS is achieved by using only half of the surviving mutants, and the improvement in MS stalls after a certain iteration of the augmentation step for different LLMs. For example, when using Codex as LLMC, in *zero-shot* learning, the MS stops improving even though, on average, 27 surviving mutants (out of 226) are not utilized in the prompt augmentation step. Similarly, in *few-shot* learning, this number is equal to 24 (out of 106).

Our results for RQ2 demonstrate that test cases generated by LLMs, regardless of the prompt type, require post-processing, such as syntax correction or intended behavior repair, in order to function properly and detect bugs effectively. Also, the order of surviving mutants to augment the prompt does not significantly impact the MS gain.

Finding 2: The *Syntax Fixing* and *Intended Behavior Repair* fix up to 95.94% and 89.86% of syntax and functional errors in test cases, respectively. The prompt augmentation in *MuTAP* decreases the unintended behavior in the output of LLMs significantly (44.36% using Codex and 52.5% using llama-2-chat). Furthermore, only a small number of mutants (up to 27) do not contribute to the improvement of MS.

Table 6
Evaluation results of *Intended Behavior Repair*. The Assertion Error Rate shows the ratio of assertions with wrong behavior.

Model	Method	Prompt	Assertion Error Rate	Repaired	Not repaired
Codex	After-refining	Zero-shot	63.63%	82.21%	17.79%
		Few-shot	62.84%	85.75%	14.25%
	MuTAP	Zero-shot	19.38%	89.71%	10.29%
		Few-shot	18.36%	90.00%	10.71%
llama-2-chat	After-refining	Zero-shot	60.27%	81.80%	18.19%
		Few-shot	63.25%	86.90%	13.09%
	MuTAP	Zero-shot	23.40%	94.06%	5.94%
		Few-shot	10.75%	94.91%	5.09%

Table 7
Evaluation of killed mutants for each type of injected operator into PUTs.

			Zero-shot				Few-shot			
			Codex		llama-2-chat		Codex		llama-2-chat	
	Pynguin		*MuTAP*				*MuTAP*			
Type	Killed	Total	Killed	Total	Killed	Total	Killed	Total	Killed	Total
AOD	13 (39.39%)	33	28 (87.50%)	32	39 (86.67%)	45	32 (94.12%)	34	40 (88.89%)	45
AOR	248 (67.39%)	368	336 (91.55%)	367	410 (91.52%)	448	347 (92.53%)	375	417 (93.08%)	448
ASR	45 (60.00%)	75	60 (80.00%)	75	79 (94.05%)	84	64 (85.33%)	75	79 (94.05%)	84
BCR	2 (40.00%)	5	2 (40.00%)	5	5 (55.56%)	9	2 (40.00%)	5	6 (66.67%)	9
COD	8 (53.33%)	15	15 (100.00%)	15	16 (72.73%)	22	17 (100.00%)	17	17 (77.27%)	22
COI	130 (81.76%)	159	154 (96.86%)	159	216 (95.15%)	227	164 (98.80%)	166	218 (96.04%)	227
EHD	1 (100.00%)	1	0 (0.00%)	0	2 (100.00%)	2	1 (100.00%)	1	2 (100.00%)	2
EXS	0 (0.00%)	0	1 (100.00%)	1	1 (100.00%)	1	1 (100.00%)	1	1 (100.00%)	1
LCR	14 (45.16%)	31	23 (74.19%)	31	37 (86.05%)	43	27 (81.82%)	33	39 (90.70%)	43
ROR	174 (66.67%)	261	227 (87.31%)	260	316 (94.61%)	334	239 (91.57%)	261	320 (95.81%)	334
SIR	10 (33.33%)	30	23 (76.67%)	30	38 (84.44%)	45	28 (82.35%)	34	40 (88.89%)	45
Total	645 (65.95%)	978	869 (89.13%)	975	1159 (91.98%)	1260	922 (92.02%)	1002	1179 (93.57%)	1260

4.2.3. RQ3: What is the performance of MuTAP for each mutation type?

In this RQ, we evaluate the performance of *MuTAP* in different mutant types. We report the total number and number of killed mutants by each method on the *HumanEval* dataset in Table 7. We report the performance of Pynguin and *MuTAP* per mutant type to help the comparison. The total number of mutants in each type is different for each method since the number of problematic *PUTs* is not the same for all methods. The MS for each type/method indicates the ratio of killed mutants out of the total number of mutants in that type. Our findings indicate that the improvement in the effectiveness of test cases generated by *MuTAP* is distributed among different types of mutants. The diversity of mutant types is correlated to the *PUTs* in our dataset. In our dataset, Arithmetic Operator Replacement (AOR), Conditional Operator Insertion (COI), and Relational Operator Replacement (ROR) are more prevalent types. Conversely, Exception Handler Deletion (EHD) and Exception Handler Swallowing (EXS) are less common in our dataset (an example for each mutant type is shown in Table 1). Although the number of mutants in EHD and EXS categories is small, both Pynguin and MuTAP with Codex and llama-2-chat faced challenges in generating test cases to detect these types. The limitation may stem from the fact that the majority of test cases (test input/output) generated by Pynguin and MuTAP are designed to assess the standard behavior of the PUT, rather than addressing exceptional or unexpected behavior.

In general, *MuTAP* shows better or similar performance in all mutant types compared to Pynguin. Considering *ASR* as an example, *MuTAP* shows better performance on this mutant type. For example, test cases generated by Pynguin identified 45 mutants in this category while test cases generated by *MuTAP* using llama-2-chat and the *few-shot* prompt identified 79 mutants in this category (out of 84).

Additionally, for another type of mutant, Break Continue Replacement (BCR), there is no improvement in the number of killed mutants when using Codex and llama-2-chat as the LLMC for *MuTAP* with a zero-shot initial prompt. The number of killed mutants increases by only one additional mutant while using llama-2-chat and a few-shot initial prompt with *MuTAP*. This result also highlights the limitation of the *MuTAP* in leveraging this type of mutant to improve the effectiveness of test cases in detecting them in the prompt augmentation step. For other types of mutants, such as CIR or LCR, *MuTAP* demonstrates significant improvement compared to Pynguin, with (33.33% vs. 88.89%) and (45.16% vs.90.70%), respectively.

Finding 3: All the different types of mutants contribute to enhancing the effectiveness of test cases generated by *MuTAP*. However, augmenting the initial prompt with several less frequent types of mutants in *MuTAP*, such as EHD, EXS, and BCR, does not result in a significant increase in the number of killed mutants compared to Pynguin.

5. Discussion

5.1. Automatic test case generation

MuTAP leverages the code synthesis capabilities of LLMs and employs prompt-based learning to assist developers in generating effective test cases without the need for the computationally expensive fine-tuning of LLMs.

LLMs are able to generate test cases that are more effective than those generated by Pynguin in terms of revealing bugs. Listing 5 shows a sample test case generated by Pynguin for the *PUT* of our Motivating example in Section 2. While Pynguin generates the test case shown in Listing 5 by creating test inputs as random integers and mutating those values to generate new test cases, LLMs produce test cases such as those in Listing 6 that are more natural-looking and correlated with input/output type and the functionality of the *PUT*. However, test cases generated by LLMs require post-processing to become more effective in detecting bugs. Our results show that augmenting the prompt with surviving mutants and refining test cases (syntax and intended behavior) helps LLMs generate more effective test cases in terms of fault detection.

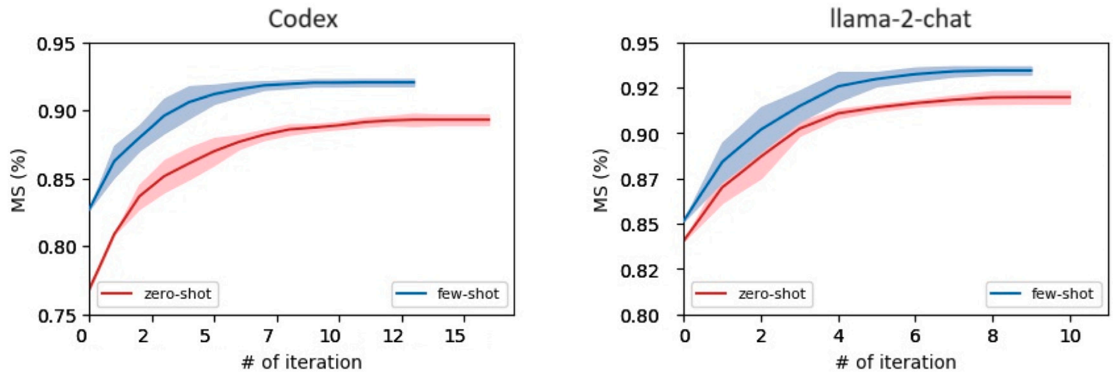

Fig. 3. The impact of using surviving mutants in different random orders on the MS. Results are averaged over 5 runs, the line shows the mean and the shaded area shows standard error. Each data point represents the average MS for all PUTs across five different runs in an iteration, wherein the surviving mutants were randomly selected for the prompt augmentation process.

Regarding the problematic PUTs, in the LLM baseline (before_refining), the presence of problematic PUTs may be attributed to both syntax and behavioral errors in the model's output. In *MuTAP*, for instance, when it employs llama-2-chat as its LLMC and applies the refining step, it successfully generates test cases for all PUTs, showing the model's ability to handle diverse constructs. Conversely, for Pynguin, its limitation in handling PUTs incorporating specific constructs, such as generators and iterators [41], potentially influences the number of problematic PUTs. In calculating the average MS, we have already excluded PUTs for which Pynguin, baselines, and *MuTAP* could not generate test cases. However, this limitation is reflected in the number of killed mutants by reporting the number of killed mutants, in comparison to the total number of mutants (1260), for both PUTs with test cases and those without. We contend that reporting various metrics (percent of problematic PUTs, MS, number of killed mutants, and percent of tasks with MS=100%) offers a fair comparison between different methods, highlighting the effectiveness of LLMs in test case generation for PUTs with different constructs. This is in contrast to a sophisticated tool like Pynguin, which, despite its intricate algorithm, encounters difficulties in handling specific programming constructs that leave a different number of PUTs without test cases.

Developers can use *MuTAP* to generate effective test cases in terms of fault detection, with the help of LLMs. Additionally, *MuTAP* can be integrated into the test generation component of the GitHub Copilot lab [53] to suggest more effective test cases for developers. Since the mutants can be generated automatically, prompt augmentation can be applied without human engagement.

```
def test_case_0():
    int_0 = -2973
    int_1 = 815
    bool_0 = module_0.any_int(int_0, int_0,
    int_1)
    assert bool_0 is False
```

Listing 5: A sample test case generated by Pynguin for the PUT in the Motivating example presented in Fig. 1.

5.2. Prompting

In the zero-shot initial prompt, one of the fixed instructions, INS_2, indicates a fixed function name for the test function. The primary reason behind using the name *"def test()"* in the initial Zero-shot prompt for the test function is twofold. Firstly, it serves as a hint to the model, indicating our expectation for a set of test cases organized within a function. Secondly, it facilitates the automation of experiments and provides an identifier or tag for automatically collecting the entire test function from the output generated by the model. Despite this, given that we already incorporate the PUT with its name in the initial prompt, it signals the model to generate a test function specifically for the mentioned PUT. In some instances, we have observed that the model (i.e., llama-2-chat) even replaced the simple test function name provided in the prompt with a more readable name in the form of "test_PUT_Name()", such as "test_any_int()". To evaluate the impact of allowing LLM to determine the function name, we randomly sampled 20% of the programming tasks from the HumanEval dataset, resulting in 33 sample cases. We omitted the test function name from the initial zero-shot prompt in INS_2 and subsequently executed it using "llama-2-chat" for these sample PUTs and applied a Mann–Whitney U-test with a significance level of 5% (alpha=0.05). The result of the test showed no significant difference in the MS or the effectiveness of test cases following this modification in the initial prompt (MS = 86.22% and standard deviation = 19.34% on the sample-set). While this preliminary analysis shows that using a predefined name as a test function name in the prompt does not affect the effectiveness of test cases in revealing bugs, the identifiers suggested by LLM are more readable and relevant to the functionality of PUTs. However, a comprehensive study involving human subjects is needed to validate this result.

To address syntax errors in test cases through re-prompting the LLMC, we used a fixed instruction (INS_fix), similar to the approach proposed by Zhang et al. [35], to leverage the LLM for syntax error fixing. If the model fails to fix the syntax error, MuTAP omits the line containing the error, based on the suggestion provided by Lemieux et al. [16]. This is because our observations indicate that the primary cause of syntax errors in the generated test cases by the model is often the last line of the test function when the model is unable to complete it. We observed that re-prompting the LLMC to address syntax errors sometimes resolves the error line by completing the last line of the test function but it introduces new lines, with one of them remaining incomplete. This primarily contributes to the lower rate of syntax error repairs by the LLM in our results. Consequently, we finalize our syntax fixing steps by omitting the error line (i.e., the incomplete line in the test functions).

In addition, for INS_3 and INS_4, the prompt serves as an instruction to call the LLMC to generate a test case capable of killing the surviving mutant in the prompt. As illustrated in Listing 6, it demonstrates the ability of LLMC to explain the differences between the surviving mutant and the PUT, pinpointing the specific line containing the bug, and subsequently producing a test case to detect that difference. However, including information about the buggy line of the mutant in the prompt could potentially enhance the effectiveness of the prompt augmentation step.

5.3. Execution time

The open-access API of Codex has a limit on the number of requests (20 per minute) and the number of tokens (40,000 per minute). For this reason, our experiment needs to stop calling the API once in a while to not exceed the limit. As a result, we present the processing time analysis using llama-2-chat. The overall processing time of *MuTAP* on *HumanEval* dataset while using llama-2-chat is on average 39.75 s with zero-shot learning (with a min of 16.16 and a max of 56.66 s) and 42.11 s with the few-shot prompt (with a min of 18.2 and a max of 64.2 s) per task. It includes on average building and calling initial prompts on LLMC with an average of 10.26 s, syntax fixing including calling the syntax fixing prompt on LLMC with 10.3 s, intended behavior repair at 0.38 s, MS calculation at 1.7 s, creating augmented prompts and calling them on LLM with 12.05 s and greedy optimization with 1.4 s. It is noteworthy that following the prompt augmentation step, *MuTAP* must reiterate the processes of syntax fixing, intended behavior repair, and greedy steps which are already included in the overall processing time. Among all steps of *MuTAP*, the most time-consuming ones are those that entail inferring output from the LLM. Conversely, the overall processing time on the same dataset with Pynguin to complete searching the required space is on average 44.16 s with a min of 2.7 and a max of 10 min which is the default timeout of the tool.

5.4. The benefit of dialog LLM

Our findings indicate that the dialog setup of llama-2-chat provides *MuTAP* with the flexibility to assign distinct roles to each component of the augmented prompt. For instance, by assigning *IUT* to an assistant role during the prompt augmentation process, the likelihood of repeating the initial tests in the generated output is reduced, while the chance of generating new test cases for detecting surviving mutants is increased. Listing 6 illustrates an example of how llama-2-chat effectively synthesizes the difference of *PUT* and one of its surviving mutants, explains the differences, and subsequently generates a new test case to detect the fault.

```
############
//prompt
############
{"role": "system", "content": "You are a Python
coding assistant. Always answer with Python code."},
{"role": "user", "content":
#Generate test cases for the following code:
‘‘‘python
def choose_num(x, y):
    if x > y:
        return -1
    if y % 2 == 0:
        return y
    if x == y:
        return -1
    return y - 1
‘‘‘
},

{"role": "assistant", "content":
‘‘‘python
#test cases
def test():
     assert choose_num(6,10) == 10
     assert choose_num(11,6) == -1
     assert choose_num(8,12) == 12
‘‘‘
},

{"role": "user", "content":
#The test function, test(), cannot detect the fault in the following code:
‘‘‘python
def choose_num(x, y):

    if x > y:
        return -1
    if y % 2 == 0:
        return y
    if x != y:
        return -1
    return y - 1
‘‘‘
#Provide a new test case to detect the fault in the prior code:
‘‘‘python
#test case
def test():
    assert
}

##################################
//Generated output of llama-2-chat
##################################
"Here is a new test case to test the fault in the previous code:"
choose_num(5, 5) == -1

‘‘‘python
def test():
     assert choose_num(5, 5) == -1
‘‘‘
"This test case checks for the scenario where both x and y are equal. In the modified
code, the condition
if x != y will
evaluate to False, and the expected
result is -1. However, in the original
code, the condition if x == y will
evaluate to True, and
the incorrect
result would be y - 1, which is 4
instead of -1."
```

Listing 6: A sample of augmented prompt and inference output on llama-2-chat.

5.5. Evaluation metrics

Prior studies [18,25,37] that involve the generation of assertions through LLMs have employed the *"exact match"* as one of their evaluation metrics. Exact match calculates the percentage of test cases generated by LLMs (the inferred output) that lexically match with the ground truth test cases (expected output). However, CIDAR [37] has already discussed the inadequacy of exact match as a suitable metric for assessing assertions produced by LLMs. This reason is that the model often generates assertions that are semantically correct but may not precisely match the ground truth. In our study, *MuTAP* executed each test case including assertions, both on the *PUT* and on its mutants to assess their correctness and effectiveness, reporting their MS. MS is a metric frequently used in prior studies and it serves as an effective metric for evaluating the quality of the test oracle [3]. While, in this paper, we focus on improving the effectiveness of test cases in terms of fault detection, there are other metrics such as test coverage that can assess other quality aspects of a test case. Improving MS does not necessarily lead to good coverage and test coverage is weakly correlated with the efficiency of tests in fault detection [26] and is challenged as a measure of test effectiveness in revealing faults [27,28], which can make it challenging for our proposed method to perform well on both metrics [30,43].

Furthermore, other researchers reported that approximately 60% of the test cases generated by Codex encounter compilation issues due to syntax errors [54]. The incorporation of syntax correction and intended behavior repair steps in our proposed method, *MuTAP*, significantly enhances the utility of the tests generated by LLMs.

5.6. Surviving mutants

We augment the prompt at each iteration for each *PUT* with a single surviving mutant. The average number of mutants for all *PUT*s in *HumanEval* and *Refactory* are 6.6 and 4.2 and the average number of surviving mutants are 3.6 and 1.8, respectively. Using a combination of surviving mutants to augment the prompt could impact the speed of reaching 100% MS. However, not all surviving mutants used in prompt augmentation contribute to improving MS, sometimes new test cases that address one mutant can also kill the remaining surviving mutants.

6. Threats to validity

Internal validity.

In this study, we employed two different prompt-based learning techniques: *zero-shot* and *few-shot*. However, we did not explore the potential impact of altering the natural language instructions or demonstrative examples (for *few-shot* learning) within our prompts. Modifying these instructions or utilizing different demonstrative examples more closely aligned with the *PUT*'s functionality could potentially enhance the results. As demonstrated by our results in RQ2, including the IUT in the prompt during augmentation steps reduced the instances of unintended behavior in test oracles. Conversely, using, for example, lengthy natural language instructions might potentially have an adverse effect on the results.

We did not integrate additional information about the syntax error in the IUT or the injected bug in the mutants, such as error messages or error lines, into the prompt. It is worth considering that including additional details about the errors or bugs may enhance the LLMC's performance to repair its initial output.

Additionally, we acknowledge that the greedy algorithm employed in our approach to minimize the number of test oracles might not be the most optimal solution for minimizing test oracles while maximizing MS. However, prior studies [9,43] using the same method to minimize the number of assertions have demonstrated its effectiveness in reducing the number of test oracles within test cases, along with its ease of implementation.

Finally, among different types of assertions, we only focus on generating primitive ones in this study. Other assertion types can be explored in future studies.

Construct Validity. We employ the notions of mutant killability and bug detection as metrics to gauge the effectiveness of test cases, given that the primary objective of testing is to uncover bugs. Coverage has been employed in various other studies to assess test case quality [15,16]. However, it has been demonstrated that while there exists a correlation between coverage and bug detection, they may not consistently align in ranking different testing strategies, as observed in the realm of fuzz testing [55].

It is important to note that the bugs present in mutants are artificial and might not directly correspond to real-world faults. To address this concern, we have employed the *Refactory* [31] dataset, a bug-repairing dataset that contains real faulty programs developed by students.

External Validity. For our experiments, we used two datasets containing Python programming tasks, which could potentially pose external challenges to the validity of our findings. The requirement for executable Python programs is essential to run the generated tests against both the accurate and buggy versions (real or mutated) of *PUT* and this consideration guided our choice of datasets. However, since we did not make any specific assumptions while selecting the dataset, our results can be extended to other Python programs.

MuTAP, like some other well-known test case generation techniques [9,41], operates under the assumption that the PUT is not buggy. However, this assumption can introduce limitations in generating test cases when the correct PUT is not accessible. Future studies can focus on leveraging LLM to generate test cases even in the presence of bugs in PUTs.

Finally, it should be acknowledged that the technique proposed and the evaluations conducted in this paper are conceptually adaptable to languages beyond Python. However, the current implementation of *MuTAP* is tailored for Python programs, meaning our existing results cannot be extended to cover other programming languages.

Reliability validity. For the purpose of enabling other researchers to replicate or expand upon our study, we provide a replication package [32]. However, the ongoing enhancement of LLMs could potentially pose a challenge to achieving an exact replication of our results.

7. Related work

Bareißet al. [12] studied the impact of *few-shot* learning across various downstream tasks, including test case and test oracle generation. They compared the performance of *few-shot* learning with automatic test generation tools. The investigation was conducted on a different set of Java methods sourced from different benchmarks. The outcomes indicated that LLMs possess the capability to generate test cases and test oracles that exactly match (in lexical terms) the ground truth tests within the benchmark projects. Furthermore, their test coverage was found to be comparable with test cases generated by automatic test generation tools.

Sch"afer et al. [15] undertook an effort to generate test cases by prompting Codex. Their investigation was concentrated on 25 JavaScript packages. The prompt in their study encompassed the implementation of the PUT and also the usage examples of APIs extracted from documentation. In instances where a test case proved unsuccessful on the PUT, their method incorporated the encountered error message into the prompt and re-prompted Codex. Their findings demonstrated that the process of enhancing the prompt with such additional information facilitated Codex in producing correct test cases with sufficient coverage.

LIBRO [56] used the issue reports (both title and body) as *few-shot* prompts to generate bug-reproducing test cases. The final test cases were incorporated into appropriate test classes and ranked based on their validity. The results revealed an enhancement in generating correct test cases to reproduce bugs compared to state-of-the-art tools.

CEDAR [37], rather than employing fixed demonstrative examples in *few-shot* learning, aimed to retrieve demonstrative examples related to each *PUT* and incorporate them into the prompt. They assessed their method based on the lexical match, termed "exact match", between generated assertions and the ground truth in a benchmark. While their proposed approach demonstrates enhanced performance in achieving exact matches between assertions and the ground truth, it necessitates an extensive pull of code samples for the selection of appropriate demonstrative examples for each *PUT*.

ATHENATEST [22] employed the BART transformer model [23], which they fine-tuned using a collection of Java functions and their corresponding tests. They reported test coverage comparable to those of EvoSuite [9] upon evaluating generating test cases for five Java projects.

TOGA [18] engaged in fine-tuning CodeBERT using the *PUT*'s docstring along with the prefix of a test case featuring a masked assertion. Their goal was to synthesize the assertion. Subsequently, they formulated the whole test oracles by incorporating a test oracle grammar and generating a set of assertions. This set was then subjected to ranking through a neural network ranker based on their lexical match to ground truth test oracles. Although they reported results akin to those of EvoSuite [9] in bug detection, their focus is only on synthesizing the assertions. However, synthesizing assertion is not challenging but generating effective and meaningful test oracles poses a significant challenge.

CODAMOSA combined the test cases generated by Codex with those derived from Pynguin in cases where Pynguin's test case generation halted and failed to enhance test coverage. CODAMOSA achieves higher test coverage on various Python benchmarks [16] compared to Pynguin. It is worth noting that, akin to other studies, CODAMOSA concentrated solely on test coverage improvement, and its generated test cases lacked assertion oracles for bug detection within programs.

Two additional studies employed Codex to simultaneously generate code and corresponding test cases based on a given problem description. Subsequently, they used these test cases to filter out buggy suggestions produced by Codex [13,14]. For code generation, they employed the problem description as a prompt, and for test case generation, they used the same problem description along with the *PUT* and a natural language instruction.

Although prior research has explored diverse strategies for generating test cases using LLMs like Codex and assessed them in terms of test coverage or lexical match with ground truth tests, none of these studies specifically focused on leveraging MT to enhance the effectiveness of the generated test cases.

8. Conclusion

In this paper, we proposed *MuTAP* as a means of improving and evaluating the ability of pre-trained LLMs to generate effective test cases. *MuTAP* first prompts its LLMC to generate test cases using *zero-shot* and *few-shot* learning. After identifying and correcting any potential syntax and return value errors in the generated test cases, *MuTAP* evaluates their effectiveness by conducting MT. Then, it uses the surviving mutants of each *PUT*, if any, as well as the initial inadequate test case to augment the initial prompt. It re-prompts its LLMC using the augmented prompt to regenerate new test cases that are capable of detecting surviving mutants.

We assessed the effectiveness of the test cases generated by LLMs to identify bugs in real and synthetic buggy programs. On average, test cases generated by *MuTAP* successfully identify 86.72% of buggy code in a bug repairing benchmark when using the LLM designed for code generation, Codex. When employing LLM with the dialog setup, llama-2-chat, *MuTAP* further improves its performance, detecting 94.06% of the buggy code, outperforming both an automatic test generation tool and a *zero-shot* and *few-shot* learning technique on LLMs. This underscores the advantage of employing LLMs as the core of an automatic test generation tool, as conventional automatic generation tools such as Pynguin lack access to the insights embedded in surviving mutants.

Although the current version of *MuTAP* employs two different LLMs to generate test cases for Python programs, its design and evaluation methodology are fundamentally adaptable to various programming languages and models. Therefore, as future work, it can be easily expanded to encompass other programming languages or incorporate new LLMs. Future studies can also explore the relationship between modifying the structure and specifics in the prompt and how this could potentially impact the effectiveness of the generated unit tests.

CRediT authorship contribution statement

Arghavan Moradi Dakhel: Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Amin Nikanjam:** Writing – review & editing, Writing – original draft, Validation. **Vahid Majdinasab:** Validation, Conceptualization. **Foutse Khomh:** Writing – review & editing, Supervision, Conceptualization. **Michel C. Desmarais:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Replication package link: https://github.com/ExpertiseModel/MuTAP.

Acknowledgments

This work is partially supported by the Canada Research Chair, Fonds de Recherche du Québec (FRQ), the Canadian Institute for Advanced Research (CIFAR), and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Appendix

To compare the effectiveness of the *MuTAP* to other methods on a PUT level, we employ the Vargha–Delaney $\hat{A}_{12}$ effect size [57] similar to previous studies that applied effect size on a PUT level [41,48], and calculate the effect size metric on the MS of unit tests generated in 10 runs per PUT with each method.

Fig. 4 presents the distribution of Vargha–Delaney $\hat{A}_{12}$ effect size [57] for different methods on the MS of unit tests generated in 10 runs per PUTs in the HumanEval dataset, all compared to *MuTAP* using llama-2-chat as its best performing LLMC. We compared the effect size of MS on the PUT level between *MuTAP* with its best-performing configuration LLMC (llama-2-chat) and other comparable methods in our study. The following numbers report where *MuTAP* (using llama-2-chat) performs better ($\hat{A}_{12} > 0.5$) or worse ($\hat{A}_{12} < 0.5$) than other comparable methods. In this comparison, we excluded the improvement achieved by *MuTAP* over problematic PUTs of other comparable methods.

MuTAP with zero-shot performed better than Pynguin on 61 PUTs and worse on 20 PUTs. *MuTAP* with the same setup performed better than before-refining on 55 PUTs and worse on 0 PUTs, while it works better than after-refining on 29 PUTs and worse on 0 PUTs. When *MuTAP* used llama-2-chat as its LLMC with a few-shot initial prompt, it performed better than Pynguin on 66 PUTs and worse on 17 PUTs. *MuTAP* with few-shot works better than before-refining on 63 PUTs, better than after-refining on 40 PUTs, and worse on 0 PUTs for both

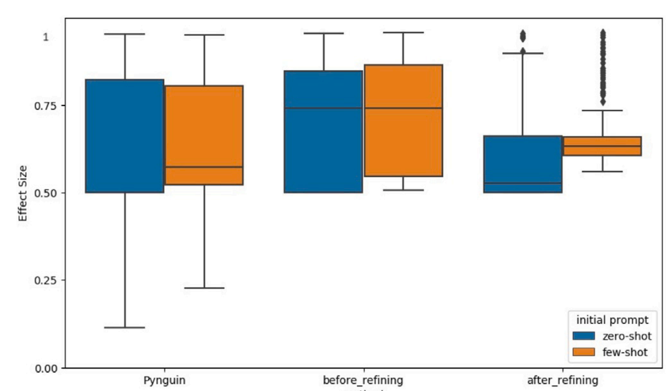

Fig. 4. Distribution of the effect size ($\hat{A}_{12}$) for each PUT on MS values presents across various methods in comparison to MuTAP, using its best performing LLMC, llama-2-chat. $\hat{A}_{12} > 0.5$ signifies that MuTAP outperforms the other alternative method, whereas $\hat{A}_{12} < 0.5$ indicates the opposite. $\hat{A}_{12} = 0.5$ indicates that there is no statistical difference between MuTAP and the alternative method.

comparable methods. The reason behind having no PUT where *MuTAP* performs worse than before-refining and after-refining is attributed to *MuTAP* improving the test cases in before-refining and after-refining by applying the augmentation step and adding more effective test cases in revealing bugs into the initial test cases. Thus, test cases generated by *MuTAP* do not perform worse than the initial test cases generated by LLMC.

References

[1] J. Shore, S. Warden, The Art of Agile Development, second ed., "O'Reilly", 2021.
[2] S. Siddiqui, Learning Test-Driven Development, "O'Reilly", 2021.
[3] T. Xie, Augmenting automatically generated unit-test suites with regression oracle checking, in: ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20, Springer, 2006, pp. 380–403.
[4] M. Selakovic, M. Pradel, R. Karim, F. Tip, Test generation for higher-order functions in dynamic languages, Proc. ACM Programm. Lang. 2 (OOPSLA) (2018) 1–27.
[5] E. Arteca, S. Harner, M. Pradel, F. Tip, Nessie: automatically testing JavaScript APIs with asynchronous callbacks, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1494–1505.
[6] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, ACM SIGSOFT Softw. Eng. Notes 30 (5) (2005) 263–272.
[7] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005, pp. 213–223.
[8] G. Fraser, A. Arcuri, Evolutionary generation of whole test suites, in: 2011 11th International Conference on Quality Software, IEEE, 2011, pp. 31–40.
[9] G. Fraser, 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, 2011, pp. 416–419.
[10] A. Panichella, S. Panichella, G. Fraser, A.A. Sawant, V.J. Hellendoorn, Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2020, pp. 523–533.
[11] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, A. De Lucia, On the diffusion of test smells in automatically generated test code: An empirical study, in: Proceedings of the 9th International Workshop on Search-Based Software Testing, 2016, pp. 5–14.
[12] P. Bareiß, B. Souza, M. d'Amorim, M. Pradel, Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code, 2022, arXiv preprint arXiv:2206.01335.
[13] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, W. Chen, Codet: Code generation with generated tests, 2022, arXiv preprint arXiv:2207.10397.
[14] S.K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J.P. Inala, C. Wang, J. Gao, Interactive code generation via test-driven user-intent formalization, 2022, arXiv preprint arXiv:2208.05950.
[15] M. Schäfer, S. Nadi, A. Eghbali, F. Tip, Adaptive test generation using a large language model, 2023, arXiv preprint arXiv:2302.06527.
[16] C. Lemieux, J.P. Inala, S.K. Lahiri, S. Sen, CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models, in: Accepted By 45th International Conference on Software Engineering, ICSE, 2023.
[17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d.O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, 2021, arXiv preprint arXiv:2107.03374.
[18] E. Dinella, G. Ryan, T. Mytkowicz, S.K. Lahiri, TOGA: a neural method for test oracle generation, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2130–2141.
[19] C.B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, N. Sundaresan, PyMT5: multi-mode translation of natural language and python code with transformers, 2020, arXiv preprint arXiv:2010.03150.
[20] M. Tufano, D. Drain, A. Svyatkovskiy, S.K. Deng, N. Sundaresan, Unit test case generation with transformers and focal context, 2020, arXiv preprint arXiv:2009.05617.
[21] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M.C. Desmarais, Z.M.J. Jiang, GitHub copilot AI pair programmer: Asset or liability? J. Syst. Softw. 203 (2023) 111734, http://dx.doi.org/10.1016/j.jss.2023.111734.
[22] M. Tufano, D. Drain, A. Svyatkovskiy, S.K. Deng, N. Sundaresan, Unit test case generation with transformers and focal context, 2021, arXiv preprint arXiv:2009.05617.
[23] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, L. Zettlemoyer, Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019, arXiv preprint arXiv:1910.13461.
[24] S. Lukasczyk, F. Kroiß, G. Fraser, An empirical study of automated unit test generation for python, Empir. Softw. Eng. 28 (2) (2023) 36.
[25] M. Tufano, D. Drain, A. Svyatkovskiy, N. Sundaresan, Generating accurate assert statements for unit test cases using pretrained transformers, in: Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, 2022, pp. 54–64.
[26] X. Cai, M.R. Lyu, The effect of code coverage on fault detection under different testing profiles, in: Proceedings of the 1st International Workshop on Advances in Model-Based Testing, in: A-MOST '05, ACM, New York, NY, USA, 2005, pp. 1–7, URL https://doi.org/10.1145/1083274.1083288.
[27] R. Gopinath, C. Jensen, A. Groce, Code coverage for suite evaluation by developers, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 72–82.
[28] H. Hemmati, How effective are code coverage criteria? in: 2015 IEEE International Conference on Software Quality, Reliability and Security, IEEE, 2015, pp. 151–156.
[29] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678, http://dx.doi.org/10.1109/TSE.2010.62.
[30] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey, in: Advances in Computers, volume 112, Elsevier, 2019, pp. 275–378.
[31] Y. Hu, U.Z. Ahmed, S. Mechtaev, B. Leong, A. Roychoudhury, Re-factoring based program repair applied to programming assignments, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 388–398.
[32] A. Moradi Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, M.C. Desmarais, The replication package, 2023, https://github.com/ExpertiseModel/MuTAP.
[33] A. Arcuri, G. Fraser, Parameter tuning or default values? An empirical investigation in search-based software engineering, Empir. Softw. Eng. 18 (2013) 594–623.
[34] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, G. Neubig, Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, ACM Comput. Surv. 55 (9) (2023) 1–35.
[35] J. Zhang, J. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, G. Verbruggen, Repairing bugs in python assignments using large language models, 2022, arXiv preprint arXiv:2209.14876.
[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.
[37] N. Nashid, M. Sintaha, A. Mesbah, Retrieval-based prompt selection for code-related few-shot learning, 2023.
[38] T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Adv. Neural Inf. Process. Syst. 33 (2020) 1877–1901.
[39] T. Ahmed, P. Devanbu, Few-shot training LLMs for project-specific code-summarization, 2022, arXiv preprint arXiv:2207.04237.
[40] H. Joshi, J. Cambronero, S. Gulwani, V. Le, I. Radicek, G. Verbruggen, Repair is nearly generation: Multilingual program repair with LLMs, 2022, arXiv preprint arXiv:2208.11640.
[41] S. Lukasczyk, G. Fraser, Pynguin: Automated unit test generation for python, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 2022, pp. 168–172.
[42] K. Hałas, Mutpy: a mutation testing tool for python 3.x source code, 2019, https://github.com/mutpy/mutpy.
[43] G. Fraser, A. Zeller, Mutation-driven generation of unit tests and oracles, in: Proceedings of the 19th International Symposium on Software Testing and Analysis, 2010, pp. 147–158.
[44] T. Dybå, V.B. Kampenes, D.I. Sjøberg, A systematic review of statistical power in software engineering experiments, Inf. Softw. Technol. 48 (8) (2006) 745–755.
[45] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, Academic Press, 2013.
[46] A. Arcuri, Test suite generation with the many independent objective (MIO) algorithm, Inf. Softw. Technol. 104 (2018) 195–206.
[47] A. Panichella, F.M. Kifetew, P. Tonella, Reformulating branch coverage as a many-objective optimization problem, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST, IEEE, 2015, pp. 1–10.
[48] A. Panichella, F.M. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, IEEE Trans. Softw. Eng. 44 (2) (2017) 122–158.
[49] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al., Llama 2: Open foundation and fine-tuned chat models, 2023, arXiv preprint arXiv:2307.09288.
[50] D. Shrivastava, H. Larochelle, D. Tarlow, Repository-level prompt generation for large language models of code, 2022, arXiv preprint arXiv:2206.12839.
[51] Y. Hu, U.Z. Ahmed, S. Mechtaev, B. Leong, A. Roychoudhury, Refactory, 2023, https://github.com/githubhuyang/refactory.
[52] S. CASS, Top programming languages 2020, 2020, https://spectrum.ieee.org/top-programming-language-2020.
[53] I. Alvarado, I. Gazit, A. Wattenberger, GitHub copilot labs, 2023, https://githubnext.com/projects/copilot-labs/.

[54] M.L. Siddiq, J. Santos, R.H. Tanvir, N. Ulfat, F.A. Rifat, V.C. Lopes, Exploring the effectiveness of large language models in generating unit tests, 2023, arXiv preprint arXiv:2305.00418.

[55] M. Böhme, L. Szekeres, J. Metzman, On the reliability of coverage-based fuzzer benchmarking, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1621–1633.

[56] S. Kang, J. Yoon, S. Yoo, Large language models are few-shot testers: Exploring LLM-based general bug reproduction, 2022, arXiv preprint arXiv:2209.11515.

[57] A. Vargha, H.D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and wong, J. Educ. Behav. Stat. 25 (2) (2000) 101–132.