Test Oracle Automation in the Era of LLMs

FACUNDO MOLINA, IMDEA Software Institute, Spain
ALESSANDRA GORLA, IMDEA Software Institute, Spain
MARCELO D'AMORIM, North Carolina State University, USA

The effectiveness of a test suite in detecting faults highly depends on the quality of its test oracles. Large Language Models (LLMs) have demonstrated remarkable proficiency in tackling diverse software testing tasks. This paper aims to present a roadmap for future research on the use of LLMs for test oracle automation. We discuss the progress made in the field of test oracle automation before the introduction of LLMs, identifying the main limitations and weaknesses of existing techniques. Additionally, we discuss recent studies on the use of LLMs for this task, highlighting the main challenges that arise from their use, e.g., how to assess quality and usefulness of the generated oracles. We conclude with a discussion about the directions and opportunities for future research on LLM-based oracle automation.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Test Oracle Problem, Large Language Models.

ACM Reference Format:
Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. 2025. Test Oracle Automation in the Era of LLMs. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 23 pages. https://doi.org/10.1145/3715107

1 INTRODUCTION

Test oracles are an essential ingredient to the construction of effective tests [8]. Precisely automating the generation of these oracles is a very challenging problem; it is akin to guessing the intention of the developer who wrote the code. For that reason, the problem continues to attract strong interest of the research community. Various techniques have been proposed to generate various kinds of oracles that can be used in testing, including *test assertions* [18, 31, 52, 55, 78, 84], *contracts* [3, 5, 10, 19, 20, 24, 25, 49–51, 75, 79, 85] (such as pre/postconditions and invariants), and *metamorphic relations* [6, 12, 13, 26, 47, 56, 71, 90], for example. Intuitively, these approaches leverage some artifact related to the SUT (e.g., documentation, code comments, code, execution traces) and then derive oracles consistent with those artifacts. For example, TOGA [18] uses source code of a given test sequence and a focal method (i.e., method under test) to infer test assertions; MeMo [12] extracts metamorphic relations by analyzing natural language comments in the source code; Daikon [20] and related tools [49, 51, 75] observe the behavior of the SUT from execution traces (e.g., traces obtained with the execution of tests) to infer likely invariants, e.g., class invariants and pre- and post-conditions. Despite all these efforts, the quest to generate tight approximations of the ground-truth oracle remains. As recently pointed by Hossain et al. [32] in a critical study about TOGA [18], more work is needed to increase precision and recall of existing approaches.

Large Language Models (LLMs) have been revolutionizing Artificial Intelligence (AI) recently. LLMs are trained on immense amounts of data and achieve impressive results in several domains.

Authors' addresses: Facundo Molina, IMDEA Software Institute, Madrid, Spain; Alessandra Gorla, IMDEA Software Institute, Madrid, Spain; Marcelo d'Amorim, North Carolina State University, Raleigh, USA.

ACM 1049-331X/2025/1-ART1
https://doi.org/10.1145/3715107

LLMs have shown useful to solve various problems in software testing [82], including automated program repair [21, 36, 87] and automated test generation [42, 63, 66]. Researchers have also started to explore the use of LLMs to generate test oracles, mainly in the form of test assertions [31, 53, 55, 78]. Although the initial results are promising, showing that generated assertions can improve test coverage [78] and some lexical/functional metrics [55], there are still aspects of LLM-generated oracles that need to be further explored.

This paper presents a roadmap to address the most significant challenges and opportunities that arise from the use of LLMs for this task. Concretely, we make the following contributions:

- we review the literature on test oracle automation before the introduction of LLMs, covering the generation of oracles that go beyond test assertions, such as contracts or metamorphic relations, identifying the challenges that remain important (Section 2);
- we review the recent advances on the use of LLMs for oracle generation, discussing their main contributions in terms of the types of oracles they generate, how they use the LLMs, and the primary sources of information employed to interact with the models (Section 3);
- we identify the most important challenges that arise from the use of LLMs for the specific task of oracle automation (Section 4);
- we share future directions and opportunities that the software engineering community should consider to address the identified challenges, exploiting the potential of LLMs for automatically deriving oracles (Section 5).

Among the main challenges that arise from the use of LLMs for test oracle automation, we include the need of more precise evaluation metrics and practices to assess the quality and usefulness (in terms of bug finding capabilities) of the generated oracles, as well as challenges related to the data used to interact with the LLMs. We also discuss how inherent challenges of LLMs, such as data leakage, dataset bias, and reproducibility, can impact LLM-based oracle generation.

Future directions and opportunities to address the identified challenges includes the development of new evaluation metrics and practices to more precisely assess oracle quality, studying and advancing bug finding capabilities of the generated oracles, and exploring novel LLM-based approaches including the use of LLM-as-a-Judge and LLM-based multi-agent approaches. Finally, we also discuss opportunities to propose novel mechanisms to interact with the LLMs and the creation of new datasets to enable the training of specialized LLMs and to mitigate dataset biases and data leakage issues. These future directions and opportunities represent promising avenues for future research and innovation in the field test oracle automation.

2 TEST ORACLE AUTOMATION BEFORE LLMS

The automated generation of test oracles has been a topic of interest in the software testing community for several years, leading to development of various approaches to automatically derive oracles either through *static* or *dynamic* techniques. Static techniques are capable of inferring oracles without executing the SUT, extracting information by analyzing the SUT's artifacts, such as the source code, comments, documentation, etc. Dynamic approaches, on the other hand, derive the oracles by observing the behavior of the SUT during execution, typically starting from a set of test cases representing concrete scenarios.

In this section, we discuss the main existing approaches for inferring oracles that do not rely on LLMs. We focus on the most common types of oracles, including test assertions, contracts and metamorphic relations. Test assertions are statements included in test cases, that check the expected behavior of the SUT in a specific scenario, and are typically expressed as code. Contracts [48] are logical constraints on a specific software element (method, class, etc.), and are usually captured in the form of pre- and post-conditions for methods, or representation invariants for classes. As

```
// Check that after the
// push operation the
// stack is not empty
@Test
public void testPush() {
  Stack s = new Stack();
  s.push(1);
  s.push(2);
  assert !s.isEmpty();
}
```

(a) Test assertion.

```
// Push an element to the stack
// Contract:
// @post: elems[size-1] = e
public void push(E e) {
  if (size == elems.length) {
    ensureCapacity();
  }
  elems[size++] = e;
  assert elems[size-1] == e;
}
```

(b) Contract (postcondition).

```
// Check the MR stating
// that pop reverts the push
// operation.
public void checkMR(Stack s1) {
  Stack s2 = s1.clone();
  s2.push(2);
  s2.pop();

  assert s1.equals(s2);
}
```

(c) Metamorphic relation.

Fig. 1. Illustration of the most common types of oracles on a simple stack example. Figure (a) shows a test assertion that checks that a stack is not empty after two push operations; Figure (b) shows a contract for the `push` method, stating that the last element pushed to the stack is the one that is expected; and Figure (c) shows a method that checks a metamorphic relation stating that the `pop` operation reverts the `push` operation for any given stack.

opposed to test assertions, contracts are not specific to a particular test case, but rather express properties that must hold for any execution. Finally, metamorphic relations express domain-specific properties of multiple executions of the SUT [68], and, compared to test assertions and contracts, they are more general oracles. Figure 1 shows examples of these three types of oracles where the SUT is a simple stack class. Figure 1(a) shows a test assertion stating that after two push operations the stack should not be empty; Figure 1(b) shows a postcondition for the push method, checking that the last pushed element is the expected one; and Figure 1(c) shows a metamorphic relation which states that the pop operation reverts the push operation.

2.1 Existing Approaches

2.1.1 Static Techniques. Several static techniques have been proposed for inferring different kinds of oracles. These approaches derive the oracles from different sources of information, such as comments accompanying the code [10, 12, 25], natural language documentation [52], or the source code itself [18, 84]. For example, Jdoctor [10, 25] is a technique that, starting from Javadoc comments uses a combination of pattern, lexical, and semantic matching to extract oracles in the form of executable procedure specifications. Other techniques such as MeMo [12] and CallMeMaybe [11] also use code comments, but for inferring metamorphic relations and method sequences respecting temporal contraints, respectively. More recently, TOGA [18] proposes a neural method that is capable of inferring assertion and exception oracles from the source code of a target test and a focal method (method under test). Assertion oracles are essentially test assertions, while exception oracles are oracles that state whether an exception is expected or not. Similarly, ATLAS [84] uses a recurrent neural network to produce assertion oracles from a test prefix and a unit under test. Differently from TOGA, ATLAS only targets test assertion oracles, without considering exceptional oracles. The main advantage of all these static techniques is that they do not require the execution of the SUT, being therefore faster and more scalable than dynamic techniques. However, they do so at the cost of accuracy. For instance, techniques based on code comments may produce inaccurate results if existing comments are not precise, which is very common due to the inherent ambiguity of natural language. Moreover, recent studies have shown that even techniques based on the state-of-the-art neural approaches have accuracy limitations, generating assertion oracles with high percentages of false positives and producing weak true positive oracles [32].

2.1.2 Dynamic Techniques. Dynamic approaches observe executions of the SUT and propose candidate oracles that are validated on the observations. Candidates oracles that are not consistent with the observations are discarded, otherwise, they are maintained and reported to the user.

Regression Assertions. The most simple dynamic techniques execute a given test case and then produce test assertions capturing some property of the execution result. This is the kind of mechanism used by mature automated test generation tools like EvoSuite [22] or Randoop [58] to incorporate test assertions oracles on the generated test cases. In the case of EvoSuite [22], the search-based approach used to generate test cases also prioritizes test assertions that maximize the detection of artificial seeded faults (mutants).

Likely Invariants. Many of the proposed dynamic techniques have focused on deriving more general types of oracles, such as contracts (like pre/postconditions or invariants). For instance, Daikon [20] is a well-known tool for inferring likely invariants using a template-based approach. Starting from a given set of test cases, Daikon maintains a set of candidate invariants that are instantiated and evaluated by executing the tests to determine the validity of the invariants on specific program points (e.g., method preconditions, method postconditions). Similarly, SpecFuzzer [49], a more recent technique built on top of Daikon, uses grammar-based fuzzing to efficiently generate thousands of candidate oracles, augmenting the set of invariants considered by the dynamic invariant detection process of Daikon. Other approaches, like EvoSpex [51] and GAssert [75], employ evolutionary computation (e.g., classic genetic algorithms or co-evolutionary algorithms) to produce candidate postcondition assertions that are again evaluated with respect to a set of test cases, usually provided by the user. Finally, machine learning techniques have also been used to capture properties typically expressed in class invariants [50, 79].

Metamorphic Relations. Dynamic techniques focusing on metamorphic relations have also employed similar strategies [6, 26, 56, 90]. For example, SBES [26] runs a search algorithm in order to discover metamorphic relations by identifying different SUT method executions that lead to the same result. More recently, MemoRIA [56] employs a combination of abstractions, grammar-based fuzzing, runtime checking and sat-based analysis to infer metamorphic relations in the form of conditionally equivalent method sequences.

2.2 Limitations and Weaknesses

Despite the significant progress made by these techniques, they still exhibit limitations and weaknesses in several aspects, that hinder their practical application.

Accuracy. This is perhaps the most critical weakness of existing techniques for oracle automation, as they often produce oracles that exhibit deficiencies. Oracle deficiencies allow to measure the quality of an oracle, by analyzing the presence of false positives and false negatives [35]. Intuitively, a *false positive* is a correct and expected program state for which the oracle is false, i.e., a false alarm. A *false negative* is an incorrect and unexpected program state for which the oracle is true, i.e., a missed fault. While false negatives are more tolerable, as one can still fix the oracle to improve its fault detection capability, false positives are more critical, as they can lead to false alarms resulting in unnecessary debugging efforts.

Though existing techniques have shown to be able to produce oracles with high fault detection capabilities, typically measured using mutation testing [60], they still exhibit high false positive rates. For instance, the static state-of-the-art neural approach TOGA can generate assertion oracles with up to 47% false positives [32]. The evaluation of dynamic techniques also shows that they often produce oracles with a significant number of false positives [3, 49, 51], representing up to 20% of the generated oracles.

Utility. Accuracy limitations have a direct impact on the utility of the oracles produced by the techniques, raising concerns about the practical usefulness of the tools [32] As existing techniques

can produce false alarms, as it is evident from their own evaluation and other reproducibility studies, it is not clear how much effort would be required by users to fix the false alarms. Moreover, the lack of studies evaluating the actual bug finding capabilities of the generated oracles, which is a more realistic measure of the utility of the oracles, poses doubts on the practical usefulness of the automatically derived oracles.

Applicability. Static techniques inferring oracles from code comments [10, 12] or natural language documentation [52] can be very efficient, but its application is limited to very well documented and mature software projects. Dynamic techniques, on the other hand, typically require a set of test cases to observe the behavior of the SUT, which can be a limitation for projects with poor or inexistent test suites. Though in these case automated test generation tools may be used to generate the required test cases, the oracle inference process becomes dependent on the quality of the generated test cases. Finally, existing techniques may also be limited on the kind of inputs they can handle (e.g., this is the case of MemoRIA [56], which is limited to stateful classes).

Expressivity. Expressiveness limitations are present in techniques that focus on inferring very specific properties (e.g., Daikon [20] through its template-based approach) or other approaches that infer oracles using specific formalisms, thus inheriting the limitations of the associated language. For instance, GAssert [75] and EvoSpex [51] use languages that target specific kinds of constraints such as logical and arithmetic constraints (without quantified expressions) in the case of GAssert and object navigation constraints (only very simple logical and arithmetic operators are supported) in the case of EvoSpex.

3 LLMS FOR ORACLE AUTOMATION

LLMs have shown a surprisingly good performance in various software engineering tasks, notably in software testing. The systematic literature review by Wang et al. [82] provides a comprehensive overview regarding how LLMs have been utilized for software testing, including the most commonly utilized LLMs, how they are used and the inputs provided to the models.

In this section, we discuss the most recent studies using LLMs to directly generate test oracles or that generate them as part of a broader test generation process. In total, we identified 37 studies that use LLMs for this purpose, including 20 that specifically address test oracle generation. All these studies are listed in Table 1. Differently from Wang et al. [82], we focus our analysis on the kinds of oracles that techniques produce, how techniques use the LLMs for the oracle generation task, and the primary sources of information employed to instruct the models to generate the oracles.

3.1 Kinds of Oracles

Existing studies using LLMs for test oracle automation focus on different types of oracles, including *unit test assertions*, *likely invariants*, *metamorphic relations*, and *exceptional oracles*. Among the 37 studies we analyzed, 29 of them (~78%) focus on unit test assertions, 3 (~8%) of the studies focus on generating likely invariants, such as pre- and postconditions and class invariants, 4 (~11%) studies generate oracles in the form of metamorphic relations, and only one study focuses on exceptional oracles.

The generation of test assertions via LLMs is the most covered type of oracle in the literature, mainly due to the fact that they are the most common type of test oracles used in software testing. There are currently 13 studies addressing the specific task of test assertion generation through the development of new LLM-based techniques [18, 28, 31, 53, 78, 81, 84], the use of state-of-the-art LLMs to assess their performance in generating test assertions [30, 40, 94], and other studies assessing the adequacy of evaluation metrics and practices [32, 45, 70]. Additionally, we found 16 studies that generate unit test assertions as part of a broader test generation process [2, 9, 16, 17, 54, 55, 57, 59, 62–64, 66, 74, 77, 91, 96], including the specific task of test completion [55].

Table 1. Collected studies using LLMs for test oracle automation.

Target Oracles	Paper	Year	Ref.
Unit test assertions	Generating Accurate Assert Statements for Unit Test Cases using Pretrained Transformers	2020	[78]
Unit test assertions	On Learning Meaningful Assert Statements for Unit Test Cases	2020	[84]
Unit test assertions	Unit Test Case Generation with Transformers	2020	[77]
Unit test assertions	TOGA: A Neural Method for Test Oracle Generation	2022	[18]
Unit test assertions	Learning Deep Semantics for Test Completion	2023	[55]
Unit test assertions	Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned	2023	[32]
Unit test assertions	An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation	2023	[66]
Unit test assertions	CAT-LM Training Language Models on Aligned Code And Tests	2023	[63]
Unit test assertions	Assessing Evaluation Metrics for Neural Test Oracle Generation	2023	[70]
Unit test assertions	Towards More Realistic Evaluation for Neural Test Oracle Generation	2023	[45]
Unit test assertions	A3Test: Assertion-Augmented Automated Test Case Generation	2023	[2]
Unit test assertions	Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools	2023	[9]
Unit test assertions	Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing	2023	[17]
Unit test assertions	Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning	2023	[53]
Unit test assertions	TOGLL: Correct and Strong Test Oracle Generation with LLMs	2024	[31]
Unit test assertions	ChatUniTest: A Framework for LLM-Based Test Generation	2024	[16]
Unit test assertions	Do LLMs generate test oracles that capture the actual or the expected program behaviour?	2024	[40]
Unit test assertions	An Empirical Study on Focal Methods in Deep-Learning-Based Approaches for Assertion Generation	2024	[30]
Unit test assertions	Multi-language Unit Test Generation using LLMs	2024	[59]
Unit test assertions	Evaluating and Improving ChatGPT for Unit Test Generation	2024	[91]
Unit test assertions	Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation	2024	[57]
Unit test assertions	Exploring Automated Assertion Generation via Large Language Models	2024	[94]
Unit test assertions	Deep Multiple Assertions Generation	2024	[81]
Unit test assertions	TDD Without Tears: Towards Test Case Generation from Requirements through Deep Reinforcement Learning	2024	[74]
Unit test assertions	Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM	2024	[64]
Unit test assertions	CoverUp: Coverage-Guided LLM-Based Test Generation	2024	[62]
Unit test assertions	CasModaTest: A Cascaded and Model-agnostic Self-directed Framework for Unit Test Generation	2024	[54]
Unit test assertions	Advancing Bug Detection in Fastjson2 with Large Language Models Driven Unit Test Generation	2024	[96]
Unit test assertions	ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance	2025	[28]
Metamorphic Relations	Can ChatGPT advance software testing intelligence? An experience report on metamorphic testing	2023	[47]
Metamorphic Relations	Automated Metamorphic-Relation Generation with ChatGPT: An Experience Report	2023	[95]
Metamorphic Relations	Towards Generating Executable Metamorphic Relations Using Large Language Models	2024	[71]
Metamorphic Relations	MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing	2024	[89]
Invariants	Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?	2023	[19]
Invariants	Can Large Language Models Reason about Program Invariants?	2023	[61]
Invariants	Impact of Large Language Models on Generating Software Specifications	2023	[88]
Exceptional oracles	Generating Exceptional Behavior Tests with Reasoning Augmented Large Language Models	2024	[93]

We identified only 8 studies inferring other types of oracles than test assertions. Among them, there are 3 studies that focus on generating likely invariants covering the use of LLMs to translate natural language intent to method postconditions [19], the generation of pre/postconditions from code comments or documentation [88], and the inference of likely invariants (including class invariants, pre/postconditions, loop invariants, etc) from source code [61]. In addition, there are 4 studies that use LLMs for deriving metamorphic relations, from requirements or system descriptions [47, 71, 95] and from previous test cases [89]. Finally, there is only one study that employs LLMs to generate exceptional behavior tests, where exceptional oracles are used [93] (i.e., oracles checking whether the method under test throws an exception or not).

3.2 LLMs Usage Strategies

As for any other software testing task, LLMs for oracle automation can be used through different strategies, which can be broadly categorized into two main groups: *pre-training and/or fine-tuning* (PT/FT) and *prompt engineering* strategies.

Pre-training and/or Fine-tuning. Pre-training entails training the model on a broad distribution of data to predict the subsequent token in a sequence. Conversely, during fine-tuning, the weights of a pre-trained model are adjusted by retraining it on a designated dataset tailored for a specific task. Prompt engineering, on the other hand, involves providing a prompt to the model that guides it to generate the desired output.

Prompt Engineering. Among the prompt engineering strategies, *zero-shot learning* and *few-shot learning* are the most common. Zero-shot learning simply involves asking the model to generate

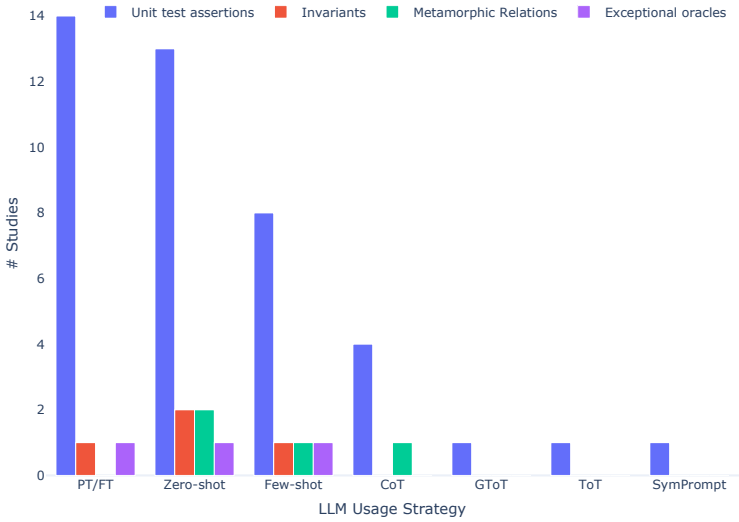

Fig. 2. Distribution of strategies employed when using LLMs to generate unit test assertions, likely invariants, metamorphic relations, and exceptional oracles. PT/FT stands for pre-training and/or fine-tuning, while all the other strategies (Zero-shot to ToT) are based on prompt engineering. Note that some studies use more than one strategy.

the results for a particular task, while few-shot learning involves prompting the model with a few examples of the task to be performed. More sophisticated prompt engineering strategies include *Chain-of-Thought* (CoT) and *Tree-of-Thoughts* (ToT). In the CoT strategy, intermediate reasoning steps are included in the prompt, while in the ToT strategy, which is built on CoT, a systematic exploration through a "tree" of reasoning steps is used, also integrating search algorithms. Though there are recent studies proposing novel prompting strategies such as *Guided Tree-of-thoughts* (GToT) [57] and *SymPrompt* [64], they are specifically designed for test generation.

Figure 2 shows the distribution of strategies employed when using LLMs to generate different types of oracles. Considering all the studies, we found 16 studies using pre-training and/or fine-tuning to generate oracles. The remaining 21 studies use some kind of prompt engineering approach, ranging from zero-shot learning to more sophisticated strategies. Below we discuss how these strategies are employed for each type of oracle.

3.2.1 Unit Test Assertions. The generation of unit test assertions with LLMs is the most studied oracle generation task in the literature. For assertion generation, pre-training and/or fine-tuning models is the more prevalent strategy. We found 14 studies using this approach, including studies that develop new models [31, 55, 78, 81, 94]. A clear example of this approach has been proposed by Tufano et al. [78], where a BART Transformer model [43] is pre-trained with a large corpus of English text and Java code, and then fine-tuned on the task of generating assert statements for unit test cases. Similarly, TeCo [55] fine-tunes the CodeT5 [83] and CodeGPT [46] LLMs specifically for the test completion task (i.e., predict the next statement in a test case), which are then evaluated for assert statement generation from the code under test (including the method under test), the test method signature, and the prior statements before the assertion statement. More recently, TOGGL [31] fine-tunes seven different code LLMs on the SF110 benchmark[23], involving various combinations of LLMs and training strategies, outperforming the neural-based approach TOGA [18].

The other 15 studies that are focused on assertion generation use the LLMs via prompt engineering. Recent improvements achieved by state-of-the-art LLMs enable the use of such models with very simple prompts. Nowadays, it is possible to use basic prompts to ask to a state-of-the-art pre-trained model, such as GPT-4 [1] or Llama2 [76], to generate test assertions for a given test case. For example, consider the test in Figure 3, which shows a very simple test case for a `Stack` class, with a unique test assertion checking that the stack is not empty after three push operations and one pop operation.

```
public void testPop() {
  Stack<Integer> stack = new Stack<>();
  stack.push(2);
  stack.push(3);
  stack.push(5);
  stack.pop();
  assertFalse(stack.isEmpty());
}
```

Fig. 3. A simple test for a `Stack` class.

If we provide the prompt "Extend the following Java test just with assert statements: + test-code", where test-code is the test `testPop`, ChatGPT with GPT-4 produces the following assertions:

```
assertEquals(5, poppedElement); // Verify the popped element
assertEquals(2, stack.size()); // Verify size after pop
assertEquals(3, (int) stack.peek()); // Verify the top of the stack is now 3
assertFalse(stack.isEmpty()); // Ensure the stack is not empty
```

checking that the popped element is 5, that the size of the stack is 2, that the top of the stack is 3, and that the stack is not empty (ChatGPT edited the test to save the result of pop in the variable `poppedElement`).

This simple zero-shot learning approach is present in several recent studies that generate assertions [40, 57, 64, 70, 74, 96]. For instance, Konstantinou et al. [40] generate test assertions with GPT-3.5 using the prompt instruction "You are a professional who writes Java test methods in JUnit4 and Java 8. Given the previous data, generate 5 possible assertions. Answer with only 5 assertions." together with other information such as the class under test, the method under test and the test prefix. Similarly, Shin et al. [70] also use GPT-3.5 through a basic query to suggest a single line oracle given a test prefix and a focal method. The complexity of the prompts can vary when using a zero-shot strategy, including the use of multiple prompts in several stages [9, 17, 59, 62, 66, 91]. This is the case of the study by Yuan et al. [91], where GPT-3.5 is utilized by first using an intention prompt that helps the model to understand the focal method under analysis, and then using a generation prompt, which instructs the model to generate a unit test containing test assertions.

The use of more elaborated prompting strategies is less frequent. The few-shot learning strategy, where the model is prompted with a few examples of the task to be performed, is used in 7 studies [17, 53, 54, 57, 59, 62, 96]. For instance, CasModaTest [54] is a model-agnostic unit test generation framework that includes an assertion generation step using a few-shot strategy. After generating a test prefix, CasModaTest prompts the LLM with the instruction "Your task now is to generate a test assertion to replace the <OraclePlaceHolder> in UNIT_TEST" and including five examples of the corresponding assertion for a given unit test and focal method signature. Though this component in CasModaTest is specifically designed for assertion generation, the majority of the studies using few-shot are intended for test generation. The study by Nashid et al. [53] is the only one that specifically targets assertion generation. They propose a technique for prompt

creation based on embedding or frequency analysis, that can achieve an exact match rate of 76% in test assertion generation.

Additionally, a few studies use more sophisticated prompting strategies. CoT is employed in 3 studies [16, 57, 96] while ToT is used in 1 study [57]. For instance, the study by Ouédraogo et al. [57], which explores these various of these prompt engineering methods to adapt LLMs for unit test generation, found that generating tests with GPT-4 using CoT seem to reduce test smells related to assertion roulette (multiple assertions included in the same test without a clear purpose). Finally, though the novel prompting strategies GToT [57] and SymPrompt [64] have been proposed in recent studies, their actual impact on the generated assertions has not been evaluated.

3.2.2 Likely Invariants. The use of LLMs for inferring likely invariants has been less explored. The 3 studies we found use some kind of prompt engineering [19, 61, 88]. For instance, Endres et al. [19] define a zero-shot prompt template to instruct chat-based LLMs (GPT-3.5, GPT-4 and StarCoder) to generate postconditions expressed as program assertions. The prompt, which asks the model to generate a symbolic postcondition consisting of exactly one assert statement, includes the focal method implementation and its natural language description. This approach is able to generate corresponding postconditions for 96% of the problems in the EvalPlus benchmark [44], and performs better than oracles generated with tools like TOGA [18] and Daikon [20]. Similarly, Xie et al. [88] employ a few-shot prompt strategy to generate pre/postconditions from code comments or documentation. In this case, the examples provided to the model are pairs of code comments and the corresponding specifications, extracted from a previous study [10].

Pei et al. [61], besides using a simple zero-shot prompting used as a baseline, proposes fine-tuning a pre-trained model to generate likely invariants. To do so, they fine-tune a Transformer model on source code tagged with likely invariants generated with the Daikon tool [20]. The model is fine-tuned to take as input the source code of a program and a target program point, and to generate the likely invariants for the state at the provided point. This approach can achieve 86% precision and recall, with better performance than Daikon, even when the set of available tests is limited.

3.2.3 Metamorphic Relations. We found 4 studies that explore the use of LLMs for generating metamorphic relations [47, 71, 89, 95]. In all these cases, LLMs are used through prompt engineering strategies. For instance, Luu et al. [47] have experimented using a two-step zero shot strategy to generate metamorphic relations with GPT-4. The authors first clarify the target system with the model, and then ask it to generate a set of unique metamorphic relations. Similarly, Zhang et al. [95] use GPT-3.5 to generate metamorphic relations for autonomous driving systems via zero-shot prompts such as "Give me five metamorphic relations (MRs) for testing the parking module of autonomous driving systems (ADSs)". These two approaches generate the metamorphic relations in natural language, which demand further processing to be transformed into executable tests.

Using LLMs to generate executable metamorphic relations can be challenging, mainly because of the domain-specific knowledge required to identify the relations, and the lack of well-defined formalisms to express them. According to the literature [68], metamorphic relations are expressed through some formalism that allows capturing the expected relationship between inputs and outputs. Metamorphic relations are typically instantiated in a test case, where a *source test* (e.g., $a1 = sort([1, 3, 2])$) is executed, then a *follow-up test* (e.g., $a2 = sort([2, 1, 3])$) is executed, and finally the relation is checked ($a1 = a2$) [69]. In this direction, Shin et al. [71] proposed a more elaborated approach. Instead of directly asking the model to generate the relations, the technique uses OpenAI's GPT-3.5 and GPT-4 via a few-shot strategy to derive executable metamorphic relations from the SUT's requirements. To achieve this, the models are queried with successive prompts that guide the model to perform simple steps, such as reading the requirements document, find the input and outputs of the system, and writing the identified metamorphic relations in a specific format. Finally,

the work of Xu et al. [89] proposes the use of LLMs to infer input transformations, which allow to reuse metamorphic relations encoded in previous test cases. While the study is not inferring new metamorphic relations, the input transformations can be used to generate follow-up tests that preserve the metamorphic relation with respect to the source test. In this case, the authors employ a Chain-of-Thought strategy, where the model is guided through a series of reasoning steps to identify source and follow-up inputs, and then generate the transformations.

A simple approach that has not been explored yet for generating executable metamorphic relations via LLMs is to take advantage of existing test cases to generate follow-up tests that preserve some relation with respect to them. For example, using the zero-shot prompt "Generate a follow-up test that is equivalent to the following test + test-code" with the `testPop` test from Figure 3, GPT-3.5 produces a test with exactly the same operations, and two additional sentences:

```
stack.push(7);
stack.pop();
```

The produced follow-up test is equivalent to the source test in the sense that both result in the same stack, i.e., if we push an element and then pop it, the stack remains the same. Using both tests, one could easily implement a new test to check the metamorphic relation, as in the example shown in Figure 1(c).

3.2.4 Exceptional Oracles. In general, the generation of exceptional oracles is less explored in the literature. We found only one study that uses LLMs to generate exceptional oracles [93] through fine-tuning. Exceptional oracles are encoded within exceptional behavior tests, i.e., tests that check that undesired events are properly captured and the adequate exceptions are thrown. In the study, the authors propose ExLong, a fine-tuned LLM that automatically generate exceptional behavior tests from a zero shot instruction prompt which includes the focal method and a target throw statement.

3.3 LLM Inputs

Figure 4 shows the various sources of information that have been used to build the prompts for the LLMs. Notably, the *focal method* code, referring to the current method under analysis for which the oracle is being generated, is the most common source of information, and is present in 30 studies (~81%). The majority of these studies (26) are focused on generating unit test assertions, while others involve the use of the focal method to generate likely invariants (2), metamorphic relations (1), and exceptional oracles (1). When inferring test assertions is very common to use as input the focal method combined with the *test prefix* [2, 18, 30, 32, 45, 55, 78, 81, 84, 94]. To a lesser extent, test assertions are generated using the focal method in combination with other sources such as the focal class [16, 40, 54, 77], the focal method signature [31, 66], or natural language documentation [9, 31, 40, 66]. For the less common types of oracles, we only found the use of the focal method together with test prefixes for generating metamorphic relations [89] and combined with documentation for inferring likely invariants [19] and exceptional oracles [93].

Other source of information that is frequently used is the test prefix, present in 19 studies (~51%). Again, most of these studies infer test assertions, while only one study generates metamorphic relations [89]. It is worth noting that the test prefix is always used in combination with other sources, primarily with the focal method, as previously discussed, but also with the focal class [40, 59, 91] and documentation [31, 40].

Natural language documentation is used as input for the LLMs in 10 studies (~27%). Similar to the focal method, documentation has been used to generate all types of oracles, with a predominance in the generation of test assertions. Among the 10 studies, 6 of them use documentation to

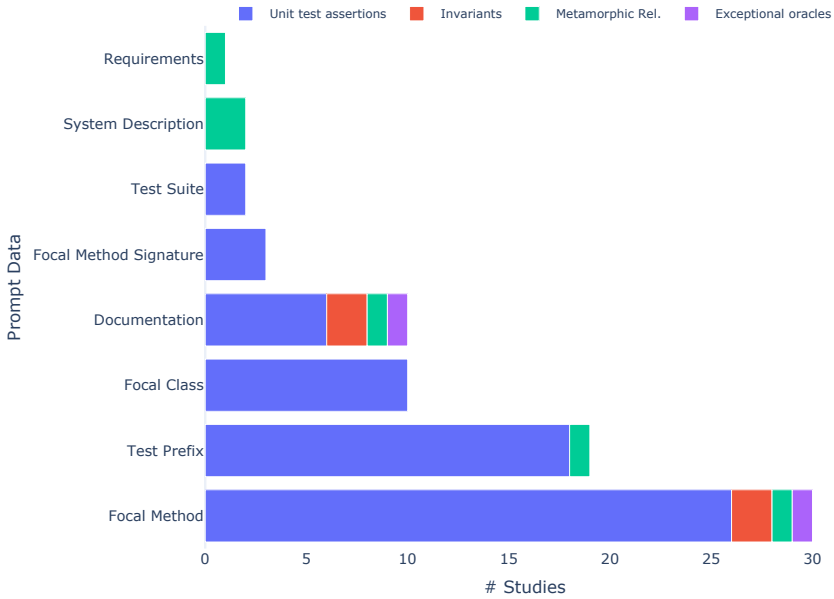

Fig. 4. Number of studies in which each source of information is used as part of the input to the LLM.

generate test assertions [9, 31, 40, 66, 74, 96], while the others focus on likely invariants [19, 88], metamorphic relations [71] and exceptional oracles [93]. Most of the time the documentation is used in combination with other sources, but there are studies that use documentation alone to generate test assertions [74], metamorphic relations [71], and likely invariants [88].

The focal class is used with a similar frequency as the documentation, also present in 10 studies (~27%). Interestingly, the focal class has only been used to infer test assertions, and in the majority of the cases is used as a source of information that complements the focal method [9, 16, 40, 54, 57, 59, 77, 91, 96]. There is, however, one study that explores the use of the focal class alone to generate tests equipped with unit test assertions [57].

Finally, a few studies use other sources of information. Some studies generate test assertions incorporating the focal method signature [31, 66] and a whole test suite [63], while others generate metamorphic relations using system descriptions [47, 95], and requirements [71]. Independently of the kind of oracle we are generating and the source of information we are using, it is evident that LLMs have an enormous potential to assist in oracle automation. However, as we discuss in the next section, the use of LLMs for this task not only inherits some limitations from previous techniques that can affect the quality of the produced oracles, but also introduce new challenges that need to be considered and addressed in future research.

4 CHALLENGES IN LLM-BASED ORACLE AUTOMATION

This section introduces the challenges in using LLMs for test oracle automation. To facilitate reference, we label each challenge with a unique identifier, having the format *C<number>*.

4.1 Oracle Quality and Usefulness

The quality and utility of the generated oracles is a critical aspect that has a direct impact on the effectiveness of the testing process.

4.1.1 C1. Oracle quality assessment. Across the studies we reviewed, the quality of the generated oracles is typically assessed by measuring accuracy in two ways: (1) exact-match rate, which is the percentage of generated oracles that exactly match the expected oracles, and (2) the success rate, which is the percentage of generated oracles that are correct with respect to a ground truth (another oracle, a test case, etc.). The former has been used in studies generating unit assertions [18, 55, 78, 84, 94], invariants [61], and exceptional oracles [93]. The latter has also been used for assessing the inference of unit assertions [31, 40], invariants [19, 88], and metamorphic relations [47, 71, 89, 95].

However, these metrics may not be enough to assess the quality of the generated oracles. The exact-match rate is a lexical metric that does not consider the semantic equivalence between the generated and expected oracles. Though this is acknowledged in most of the studies, it is important to consider semantics for more precise assessments (an oracle may be correct even if it does not exactly match the expected oracle). The success rate, on the other hand, presents the challenge of generalizability, specially for general oracles like invariants and metamorphic relations. For instance, measuring the success rate of postconditions with respect to a set of tests [19, 88] does not guarantees that the inferred postconditions are correct for all possible inputs.

4.1.2 C2. Bug detection capabilities. . The usefulness of the inferred oracles in a practical setting, such as bug finding, has not been extensively evaluated. Less than half of the studies we reviewed have conducted empirical evaluations to assess the bug finding capabilities of the generated oracles. Moreover, they typically do so through mutation analysis [60], by inserting artificial faults in the code and then checking if the generated oracles can detect them. There are only 5 studies using real world bugs to evaluate the bug finding capabilities of the generated oracles [18, 19, 81, 94, 96], where the best performing approach can detect up to 64 bugs in the Defects4J benchmark [37]. Though the mutation score (percentage of mutants detected in mutation analysis) is widely accepted as a good indicator of fault detection capabilities [38], more work is needed to understand the actual utility of LLM-generated oracles in practice. Considering that the commonly used Defects4J benchmark includes more than 800 bugs, and that the best performing approach can detect only 64 bugs [19], it is clear that the challenge of generating high-quality oracles that can effectively detect bugs in real-world software is still open.

4.2 LLMs Data

Inferring oracles via LLMs requires data either (1) to build prompts or (2) to pre-train and/or fine-tune corresponding models. The following sections discuss challenges related with the data.

4.2.1 C3. Data for prompt engineering. Zero-shot learning can be used to instruct a LLM to generate an assertion from a *test prefix* provided as context [40, 57, 70]. Few-shot learning may be necessary for more challenging tasks in oracle automation. For example, the user may need to provide *input-output examples* to "explain" the LLM how to obtain postconditions or metamorphic relations from input data [7, 19, 47, 95]. Likewise, using a chain-of-thought prompt design may be helpful to break down complex tasks in smaller tasks. For example, Hayet et al. [28] requests LLM to first summarize related *classes* before requesting the LLM to generate oracles involving those classes. Currently, there are few studies using these prompting approaches for this kind of oracles [71, 88, 89]. More research is needed to understand the impact of different prompting strategies on the quality and usefulness of this kind of oracles.

4.2.2 C4. Data for pre-training and/or fine-tuning models. Pre-training and fine-tuning can also be used to specialize the LLM for a given task and, consequently, improve its performance (Section 3 elaborates on these mechanisms). These approaches require large amounts of curated data. While for unit assertions there are large datasets available, which enabled the development of

various techniques for assertion generation [18, 55, 78, 84, 94], curated datasets for invariants or metamorphic relations are still scarce and can be challenging to obtain and maintain, specially to properly represent scenarios that require domain-specific knowledge, e.g., to generate metamorphic relations. For instance, the study by Pei et al. [61], which fine-tunes a model to generate Daikon-like invariants, opens a promising direction for the specialized use of LLMs to generate broader oracles.

4.3 Results Reproducibility

4.3.1 C5. Use of closed-source models. Most of the existing studies utilizing LLMs for oracle automation are based on closed-source models, such as OpenAI's GPT-3.5 and GPT-4. In our review, we found that 21 studies (~60%) are using some of these models, which are frequently updated or deprecated. These kind of changes on the way in which the models are accessed challenges the reproducibility of the results, as previous the results of the studies may become obsolete or impossible to replicate [65]. Notably, considering the studies we reviewed, 13 out of the 21 studies using closed-source models mitigate this reproducibility threat by incorporating in the evaluation open-source models like StarCoder, Llama2, Llama3 or CodeLlama [19, 63, 66, 88, 89]. This not only enables some level of reproducibility, but also allow comparative analysis between different kinds of models. It is also important to provide access to the defined prompts, the parameter settings used to interact with the models, and the artifacts produced by the models, in order to increase transparency and replicability.

4.3.2 C6. LLM output unpredictability. The unpredictability of LLMs outputs, i.e., the fact that the same model can produce different outputs for the same input prompts when executed multiple times, is another threat to reproducibility that should be considered. Thus, conducting multiple runs and using variability metrics to assess this aspect is also important [65].

4.4 Dataset Bias

LLMs heavily depend on a large number of huge datasets for training and fine-tuning. For instance, The Stack [39], a widely used dataset for training LLMs for software engineering tasks, contains a one trillion tokens sourced involving more than 80 programming languages, GitHub issues, Git commits, and Jupyter notebook. The quality, diversity, and quantity of these datasets can directly affect the performance and generalizability of the models. Moreover, biases in the training data can be inherited by the model, leading to biased or inaccurate predictions [33].

4.4.1 C7. Handling dataset biases. Dataset biases can also affect the quality of the oracles produced by LLM-based approaches, and may present challenges for their generalization, as well as for the automation of different kinds of oracles expressed in specific formalisms. Surprisingly, only 5 studies in our literature review either acknowledge potential dataset biases [18, 32] or take concrete actions to mitigate the issue [30, 57, 66]. For instance, Schäfer et al. [66] and Ouédraogo et al. [57] mitigate dataset biases by evaluating their approach on less popular packages and other repositories that were not included in the model's training. He et al. [30], on the other hand, propose a new way of creating more precise datasets resembling real-world practices for DL-based assertion generation, by previously revealing that existing approaches may be sensitive to focal methods. Other mechanisms to mitigate dataset biases may include the elimination of duplicated data while trying to increase the diversity and uniqueness of the data [33].

The low number of studies addressing potential dataset biases in the context of LLM-based oracle automation suggests that more research is needed to minimize the risks of biases.

```
public void testSearch() {
  final ArrayStack<E> stack = makeObject();
  stack.push((E) "First Item");
  stack.push((E) "Second Item");
-----------------------------------------------------------------
  // Test searching for existing elements
  assertEquals(2, stack.search("First Item")); // First Item is at index 2 from the top
  assertEquals(1, stack.search("Second Item")); // Second Item is at index 1 from the top

  // Test searching for non-existing element
  assertEquals(-1, stack.search("Non-existing Item")); // Non-existing Item is not found in
     the stack
```

Fig. 5. Test from Defects4J, and the assertions produced by ChatGPT-3.5.

4.5 Oracle Leakage

4.5.1 C8. Oracle leakage. Data leakage issues are particularly relevant for LLM-based oracle automation, specifically during the evaluation of these approaches. As LLMs are trained on very large amounts of data, often including publicly available code from GitHub [15], there is a risk that the model has memorized some code samples from the training data [34]. Thus, when evaluating LLM-based techniques for oracle automation, it is important to pay special attention to the data used in the evaluation of the techniques, as we may end up obtaining oracles that are not actually created by the model, but rather replicated from the training data.

To illustrate this issue, let us consider Defects4J [37], one the most widely-adopted benchmarks in software testing research. Many of the projects involved in Defects4J are publicly available on GitHub. Thus, evaluating LLM-based oracle generation techniques on Defects4J can clearly lead to oracle leakages, and make the LLM provide accurate oracles just because they are a copy of the oracles from the training data. Figure 5 shows an example of a test prefix (i.e., the first part of a test without the assertions) from the Apache Commons Collections project in Defects4J, available in revision `7c99c62` of the project repository[1], and the test assertions generated by ChatGPT-3.5. With the prompt "Complete the following Java test with test assertions: + test-code", ChatGPT-3.5 produces exactly the same assertions as in the original test case, with the sole difference that natural language messages to explain the expected behavior are included.

Compared to the dataset bias issue, the oracle leakage problem is more addressed in the literature, with near half of the studies we reviewed acknowledging or mitigating the issue. The most common action to mitigate the problem is to ensure that the evaluation data is not included in the training data [2, 40, 55, 57, 64, 77, 89, 94], often using datasets containing code produced after the models training, such as the GitBug-Java benchmark [73], which is after the cut-off date of the training data of most of the notable LLMs, including OpenAI models. Other concrete actions include normalizing method names [18], measuring the similarity or distance between the generated output and existing data [66, 91], and generating the oracles from different program versions [45]. Additionally, other actions could include the use of evaluation data from multiple sources, as recommended by Sallou et al. [65]. SourceForge projects are potentially a good source of data for evaluation, as they have been shown that LLMs can have a worse performance on them compared to GitHub projects [72].

It is worth to note that this issue is more critical when inferring unit test assertions compared to other kind of oracles, as they are more likely to be found in the training data. The concern is partially mitigated by the task itself when inferring oracles such as invariants and metamorphic

[1] https://github.com/apache/commons-collections

relations, as these kind of oracles are rarely found in source code, specifically when trying to generate them in specific and less-common formalisms.

5 FUTURE DIRECTIONS AND OPPORTUNITIES

We provide a roadmap for future research on LLM-based oracle automation, covering opportunities for addressing the main challenges and limitations of existing techniques, as well as for developing more reliable and effective LLM-based techniques.

5.1 Assessing Oracle Quality (C1)

5.1.1 Use of semantic metrics. Deriving accurate oracles is crucial for obtaining high-quality test suites, that can effectively detect faults in the SUT. LLMs have shown a great potential for extracting oracles that are accurate mainly in terms of exact-match rate and success rate, as discussed in Section 4.1. Related to these metrics, there are clear opportunities for future research to develop more precise evaluation techniques. The exact-match rate is a lexical metric that does not consider the semantic equivalence between the generated and expected oracles. Indeed, there has been shown a low correlation between textual similarity metrics and semantic metrics such as line coverage and mutation score [70]. Thus, exploring other semantic metrics such as checked coverage [67], the ability to detect the same set of bugs, or even analyze the logical equivalence between the generated and expected oracles could provide a more adequate measure of the accuracy of the LLM-produced oracles, and is an opportunity for future research.

5.1.2 Iterative refinement. The success rate is a metric measured with respect to a ground truth, which may be limited and lead to false positives or negatives. For instance, Endres et al. [19] measure the success rate with respect to a set of tests, which does not guarantee that the inferred postconditions are correct for all possible inputs. Though this is an issue that affects oracle automation techniques in general, it has not been extensively explored, being an opportunity for future research. For instance, OASIs [35], which uses evolutionary computation to search for false positives and negatives in assertion oracles, could be used in combination with LLM-based techniques to iteratively refine the oracles until no more deficiencies are detected, as done in GAssert [75]. Moreover, other oracle deficiencies detection techniques could be developed, possibly based on other metrics (e.g., checked coverage), to provide more guarantees on the quality of the oracles.

All these techniques reporting oracle deficiencies can be integrated into a pipeline in which the LLM-generated oracles are subjected to an assurance process, and iteratively improved. This is the idea behind the Assured LLM-based Software Engineering (Assured LLMSE) framework proposed by Alshahwan et al. [4], which aims to provide guarantees on the output produced by LLMs used for software engineering tasks. Figure 6 shows an overview of how Assured LLMSE could be applied to LLM-based oracle generation, where the LLM-produced oracles are subjected to an assurance process, possibly looking for oracle deficiencies, and then refined until no more deficiencies are detected. The assurance process can also provide information to re-prompt the model with those oracles that do not pass the process.

5.2 Advancing Bug Finding Capabilities (C2)

The ultimate goal of test oracle automation is to produce strong and precise oracles that can increase the bug finding capabilities during the testing process.

5.2.1 Empirical studies on bug finding. The fact that the ability of LLM-generated oracles to detect real bugs (such as those in Defects4J) has not been extensively evaluated, and the relatively low effectiveness in the best performing approach [19], opens an opportunity for future research. Large-scale empirical studies on widely used open source projects could be conducted to evaluate the

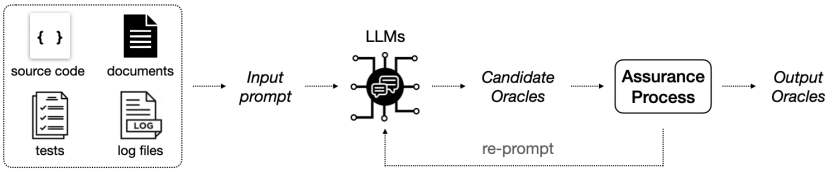

Fig. 6. A workflow for Assured LLM-based Oracle Generation.

performance of inferred oracles to find failures that reveal the presence of unexpected behavior in the SUT. Moreover, as most of the recent studies only analyze the detection of Defects4J bugs, it is important to explore other datasets, including datasets in other languages, not just Java. A recent dataset of real-world Java bugs, GitBug-Java [73], containing 199 bugs from 2023, could be used for this purpose, as it is after the cut-off date of the training data of various LLMs. Additionally, in-depth analysis of the taxonomy of bugs that LLM-generated oracles can detect could be conducted, to shed light on how to address the generation of more effective oracles. Other target datasets could include the ManyBugs and IntroClass Benchmarks [41], containing 1,183 defects in C programs, and the BugsInPy dataset [86], which includes 493 real-world Python bugs.

5.3 New LLM-based Approaches (C1 and C2)

There are also opportunities for developing new LLM-based approaches that can lead to improvements in the test oracle automation process.

5.3.1 LLMs-as-a-Judge. Konstantinou et al. [40] explore the use of LLMs to classify unit test assertions as either correct or incorrect, from a simple (zero-shot) prompt. Though the study shows that LLMs are better at generating the assertions than classifying them, the use of LLMs as a judge could be further explored, specially leveraging existing techniques (see Section 2.1) to propose candidate oracles. For instance, one could use existing evolutionary approaches to generate the candidate oracles [26, 51, 75] and then use the LLM to classify or rank them.

5.3.2 LLM-based Multi-agent approaches. Multi-agent systems are composed of multiple agents, each with its own skills and responsibilities, that work together towards a common goal. Implementing this kind of systems with LLM-based agents could benefit various software engineering tasks [29], specially test oracle automation. For instance, one agent could be responsible for generating candidates oracles, while another could be responsible for evaluating the adequacy of the candidates, or even looking for oracle deficiencies. To the best of our knowledge, there are no studies exploring this kind of approach in the context of LLM-based oracle automation.

5.3.3 Human feedback integration. Human feedback can be useful as a way to select meaningful oracles, specially when the inferred oracles are intended to be used to detect unknown bugs in non-regression settings. For instance, AutoAssert [92] implements a human-in-the-loop approach to generate assertions for test cases, where the user has the opportunity to accept, modify or delete the assertions. This technique works by employing a dynamic analysis on input variables to produce candidate assertions trying to match the developer's style in the project under analysis. Given the well-known ability of LLMs to recognize patterns in the data, they could be used to suggest candidate assertions in a similar way, allowing the user to decide which assertions are meaningful and which are not.

Furthermore, as LLMs have shown to be proficient in tasks like code summarization [27], another interesting direction that could be explored is to leverage LLMs to produce an explanation of the

SUT, so that this information can be exploited to generate more meaningful oracles, possibly by the user. Proposing effective and efficient techniques that integrate human feedback in the oracle automation process has the potential to improve the accuracy and utility of the generated oracles.

5.4 Input Data for LLMs (C3)

5.4.1 Novel prompt engineering strategies. Understanding how to design effective prompts, and exploring other strategies for prompt engineering, can lead to high quality test oracles and is an important research direction, specially given the increasing availability of LLMs with different capabilities and sizes. While straightforward zero-shot and few-shot learning approaches have been used for inferring most of the discussed oracle types, the use of more complex strategies, such as chain-of-thought, tree-of-thought, or guided tree-of-thought prompts, has mostly been explored for unit test assertions [57]. Exploring these strategies for deriving invariants or metamorphic relations, could be a promising direction.

Moreover, new prompting techniques could be developed. For instance, SymPrompt [64], a novel prompting technique designed for effective test generation, implements a multi-step process in which the model is subsequently asked to generate tests that cover different execution paths, leading to tests with higher coverage. Similarly, approaches like this could be explored to generate oracles that increasingly optimize some desired criteria, such the ability of the oracles to the detect mutants in the method under test.

5.5 New Datasets for Training and Evaluation (C4, C7 and C8)

There also opportunities for developing new datasets, that can lead enable the development of more specialized techniques through pre-training and fine-tuning, while also minimizing the risks of biases and data leakages.

5.5.1 Pre-training and fine-tuning. The availability of large and diverse datasets is crucial for developing LLM-based techniques through pre-training and fine-tuning. As previously discussed, for invariants or metamorphic relations datasets for pre-training and fine-tuning are still scarce. Pei et al. [61] fine-tune a model using a dataset equipped with Daikon-like invariants. This study opens a promising direction for creating other specialized large datasets, either for invariants or metamorphic relations, that can be used for pre-training or fine-tuning LLMs. For instance, new datasets could include invariants expressed in the Java Modeling Language (JML) [14] or postconditions expressed as assertions. Moreover, as previous studies have shown, the use of specific software engineering data (such as data from Stack Overflow, GitHub, and Jira Issues) is valuable and can lead to improvements in the performance of the models [80].

5.5.2 Dataset biases. At the same time, the development of new datasets is crucial to minimize the risks of biases. When creating new datasets is important to maximize the diversity and uniqueness of the data [33], covering different application domains and project sizes, including popular and widely-used projects, as well as less popular projects. While collecting such projects may be straightforward for unit test assertions (given their availability) it can be more challenging for other kinds of oracles, requiring a substantial effort to equip projects with the desired oracles.

5.5.3 Oracle leakage. Finally, data leakage issues, which affect LLM-based techniques in general, poses opportunities for future research on the development of new datasets for evaluating oracle automation approaches. Such datasets should be carefully curated to ensure that the data is not present in the training data of the LLMs. As suggested by Sallou et al. [65], one can mitigate data leakage issues by assessing the LLMs on metamorphic data. Following this direction, one could assess LLM-based oracle generation techniques with metamorphic testing. For instance, if we are

generating assertions (resp. invariants) from a given test prefix (resp. method), we could generate new test prefixes (resp. methods) through semantic preserving transformations, and then check if the oracles produced by the LLMs are consistent with the expected behavior. Generating evaluation data through a process like this could provide more guarantees on the quality of the produced oracles and lead to more reliable results, minimizing data leakage issues.

6 CONCLUSION

Thanks to the ability of LLMs to quickly generate content, either as code or as specifically formatted text, they have an enormous potential to improve software testing tasks. In this paper, we focus on the use of LLMs for automating the generation of test oracles. We discuss the limitations and weaknesses of the state-of-the-art techniques for oracle automation, and how LLMs can help to address these limitations. The accuracy of automatically generated oracles and their utility in practice are still aspects that need to be further explored and improved, and the use of LLMs for this task has a promising potential.

Additionally, we discuss the main challenges that arise from the use of LLMs for oracle automation, including aspects related to the assessment of the quality and usefulness of the generated oracles, the input data used to prompt the LLMs, results reproducibility, dataset bias, and data leakages. Addressing these challenges is crucial to ensure the inference of reliable and effective oracles, trough reproducible and transparent research, and to avoid oracle leakages that can lead to to the generation of oracles that are not actually created by the model, but are rather replicated from the training data. We provide considerations that SE researchers could take into account to address these challenges.

Finally, we present a roadmap for future research and opportunities for advancing the field of LLM-based oracle automation, which includes improving oracle quality assessment, addressing oracle deficiencies, studying and advancing the bug finding capabilities of the oracles, exploring new LLM-based approaches and new strategies for prompt engineering, and proposing new datasets to ensure the reliability of the results. We believe that addressing all these challenges and opportunities can lead to the development of more reliable and effective LLM-based techniques for oracle automation, with more guarantees on the quality of the produced oracles, and with a higher bug finding capability, ultimately contributing to more reliable software systems.

ACKNOWLEDGMENTS

This work is supported by the Ramón y Cajal fellowship RYC2020-030800-I and by the Spanish Government through grants TED2021-132464B-I00 (PRODIGY) and PID2022-142290OB-I00 (ESPADA). Those projects are co-funded by European Union ESF, EIE, and NextGeneration funds. This work is also funded by the National Science Foundation grant number CCF-2349961.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, and Shyamal Anadkat et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[2] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3Test: Assertion-Augmented Automated Test case generation. *Inf. Softw. Technol.* 176 (2024), 107565. https://doi.org/10.1016/J.INFSOF.2024.107565

[3] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1018–1030. https://doi.org/10.1145/3597926.3598114

[4] Nadia Alshahwan, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-Based Software Engineering. *CoRR* abs/2402.04380 (2024). https://doi.org/10.48550/ARXIV.2402.04380 arXiv:2402.04380

[5] Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485481

[6] Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella. 2024. GenMorph: Automatically Generating Metamorphic Relations via Genetic Programming. *IEEE Trans. Software Eng.* 50, 7 (2024), 1888–1900. https://doi.org/10.1109/TSE.2024.3407840

[7] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. arXiv:2206.01335 [cs.SE] https://arxiv.org/abs/2206.01335

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[9] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools. In *LLM4CODE@ICSE*. 54–61. https://doi.org/10.1145/3643795.3648396

[10] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 242–253. https://doi.org/10.1145/3213846.3213872

[11] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2022. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Compliant with Temporal Constraints. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022*. 19:1–19:11.

[12] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181 (2021), 111041. https://doi.org/10.1016/j.jss.2021.111041

[13] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. 2014. Cross-checking Oracles from Intrinsic Software Redundancy. In *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India, 931–942. https://doi.org/10.1145/2568225.2568287

[14] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 342–363. https://doi.org/10.1007/11804192_16

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, and Greg Brockman et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[16] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 572–576. https://doi.org/10.1145/3663529.3663801

[17] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained Large Language Models and mutation testing. *Inf. Softw. Technol.* 171 (2024), 107468. https://doi.org/10.1016/J.INFSOF.2024.107468

[18] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2130–2141. https://doi.org/10.1145/3510003.3510141

[19] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE (2024), 1889–1912. https://doi.org/10.1145/3660791

[20] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[21] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[22] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*. ACM, 416–419.

[23] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 8:1–8:42. https://doi.org/10.1145/2685612

[24] Aayush Garg, Renzo Degiovanni, Facundo Molina, Maxime Cordy, Nazareno Aguirre, Mike Papadakis, and Yves Le Traon. 2023. Enabling Efficient Assertion Inference. In *34th IEEE International Symposium on Software Reliability*

Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023. IEEE, 623–634. https://doi.org/10.1109/ISSRE59848.2023.00039

[25] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *ISSTA 2016: Proceedings of the 2016 International Symposium on Software Testing and Analysis*. Saarbrücken, Germany, 213–224. https://doi.org/10.1145/2931037.2931061

[26] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo Tonella. 2014. Search-based synthesis of equivalent method sequences. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 366–376. https://doi.org/10.1145/2635868.2635888

[27] Jian Gu, Pasquale Salza, and Harald C. Gall. 2022. Assemble Foundation Models for Automatic Code Summarization. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 935–946. https://doi.org/10.1109/SANER53432.2022.00112

[28] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. 2025. ChatAssert: LLM-based Test Oracle Generation with External Tools Assistance. *IEEE Transactions on Software Engineering* 51, 1 (2025), 305–319. https://doi.org/10.1109/TSE.2024.3519159

[29] Junda He, Christoph Treude, and David Lo. 2024. LLM-Based Multi-Agent Systems for Software Engineering: Vision and the Road Ahead. *CoRR* abs/2404.04834 (2024). https://doi.org/10.48550/ARXIV.2404.04834 arXiv:2404.04834

[30] Yibo He, Jiaming Huang, Hao Yu, and Tao Xie. 2024. An Empirical Study on Focal Methods in Deep-Learning-Based Approaches for Assertion Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1750–1771. https://doi.org/10.1145/3660785

[31] Soneya Binta Hossain and Matthew Dwyer. 2024. TOGLL: Correct and Strong Test Oracle Generation with LLMs. arXiv:2405.03786 [cs.SE] https://arxiv.org/abs/2405.03786

[32] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian G. Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 120–132. https://doi.org/10.1145/3611643.3616265

[33] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2024). https://doi.org/10.1145/3695988 Just Accepted.

[34] Huseyin A. Inan, Osman Ramadan, Lukas Wutschitz, Daniel Jones, Victor Rühle, James Withers, and Robert Sim. 2021. Training Data Leakage Analysis in Language Models. (2021). arXiv:2101.05405 [cs.CR]

[35] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 247–258. https://doi.org/10.1145/2931037.2931062

[36] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[37] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[38] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 654–665. https://doi.org/10.1145/2635868.2635929

[39] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. arXiv:2211.15533 [cs.CL] https://arxiv.org/abs/2211.15533

[40] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. 2024. Do LLMs generate test oracles that capture the actual or the expected program behaviour? arXiv:2410.21136 [cs.SE] https://arxiv.org/abs/2410.21136

[41] Claire Le Goues, Neal J. Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Software Eng.* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[42] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[43] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2019. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *CoRR* abs/1910.13461 (2019). arXiv:1910.13461 http://arxiv.org/abs/1910.13461

[44] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[45] Zhongxin Liu, Kui Liu, Xin Xia, and Xiaohu Yang. 2023. Towards More Realistic Evaluation for Neural Test Oracle Generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 589–600. https://doi.org/10.1145/3597926.3598080

[46] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. (2021). arXiv:2102.04664 [cs.SE]

[47] Quang-Hung Luu, Huai Liu, and Tsong Yueh Chen. 2023. Can ChatGPT advance software testing intelligence? An experience report on metamorphic testing. arXiv:2310.19204 [cs.SE] https://arxiv.org/abs/2310.19204

[48] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[49] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1008–1020. https://doi.org/10.1145/3510003.3510120

[50] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. 2019. Training binary classifiers as data structure invariants. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 759–770. https://doi.org/10.1109/ICSE.2019.00084

[51] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

[52] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise Oracles from structured natural language specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 188–199. https://doi.org/10.1109/ICSE.2019.00035

[53] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[54] Chao Ni, Xiaoya Wang, Liushan Chen, Dehai Zhao, Zhengong Cai, Shaohua Wang, and Xiaohu Yang. 2024. CasModaTest: A Cascaded and Model-agnostic Self-directed Framework for Unit Test Generation. *CoRR* abs/2406.15743 (2024). https://doi.org/10.48550/ARXIV.2406.15743 arXiv:2406.15743

[55] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2111–2123. https://doi.org/10.1109/ICSE48619.2023.00178

[56] Agustín Nolasco, Facundo Molina, Renzo Degiovanni, Alessandra Gorla, Diego Garbervetsky, Mike Papadakis, Sebastián Uchitel, Nazareno Aguirre, and Marcelo F. Frias. 2024. Abstraction-Aware Inference of Metamorphic Relations. *Proc. ACM Softw. Eng.* 1, FSE (2024), 450–472. https://doi.org/10.1145/3643747

[57] Wendkûuni C. Ouédraogo, Abdoul Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. 2024. Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation. *CoRR* abs/2407.00225 (2024). https://doi.org/10.48550/ARXIV.2407.00225 arXiv:2407.00225

[58] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 75–84. https://doi.org/10.1109/ICSE.2007.37

[59] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2024. Multi-language Unit Test Generation using LLMs. *CoRR* abs/2409.03093 (2024). https://doi.org/10.48550/ARXIV.2409.03093 arXiv:2409.03093

[60] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* 112 (2019), 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[61] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520. https://proceedings.mlr.press/v202/pei23a.html

[62] Juan Altmayer Pizzorno and Emery D. Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. *CoRR* abs/2403.16218 (2024). https://doi.org/10.48550/ARXIV.2403.16218 arXiv:2403.16218

[63] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2023. CAT-LM Training Language Models on Aligned Code And Tests. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 409–420. https://doi.org/10.1109/ASE56229.2023.00193

[64] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE (2024), 951–971. https://doi.org/10.1145/3643769

[65] June Sallou, Thomas Durieux, and Annibale Panichella. 2023. Breaking the Silence: the Threats of Using LLMs in Software Engineering. *CoRR* abs/2312.08055 (2023). https://doi.org/10.48550/ARXIV.2312.08055 arXiv:2312.08055

[66] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[67] David Schuler and Andreas Zeller. 2013. Checked coverage: an indicator for oracle quality. *Softw. Test. Verification Reliab.* 23, 7 (2013), 531–551. https://doi.org/10.1002/STVR.1497

[68] Sergio Segura, Gordon Fraser, Ana Belén Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[69] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. 2020. Metamorphic Testing: Testing the Untestable. *IEEE Softw.* 37, 3 (2020), 46–53. https://doi.org/10.1109/MS.2018.2875968

[70] Jiho Shin, Hadi Hemmati, Moshi Wei, and Song Wang. 2024. Assessing Evaluation Metrics for Neural Test Oracle Generation. *IEEE Trans. Software Eng.* 50, 9 (2024), 2337–2349. https://doi.org/10.1109/TSE.2024.3433463

[71] Seung Yeob Shin, Fabrizio Pastore, Domenico Bianculli, and Alexandra Baicoianu. 2024. Towards Generating Executable Metamorphic Relations Using Large Language Models. In *Quality of Information and Communications Technology*, Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini (Eds.). Springer Nature Switzerland, 126–141.

[72] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. (2024). arXiv:2305.00418 [cs.SE]

[73] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. (2024). arXiv:2402.02961 [cs.SE]

[74] Wannita Takerngsaksiri, Rujikorn Charakorn, Chakkrit Tantithamthavorn, and Yuan-Fang Li. 2024. TDD Without Tears: Towards Test Case Generation from Requirements through Deep Reinforcement Learning. *CoRR* abs/2401.07576 (2024). https://doi.org/10.48550/ARXIV.2401.07576 arXiv:2401.07576

[75] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1178–1189. https://doi.org/10.1145/3368089.3409758

[76] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, and Shruti Bhosale et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/arXiv.2307.09288 arXiv:2307.09288

[77] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers. *CoRR* abs/2009.05617 (2020). arXiv:2009.05617 https://arxiv.org/abs/2009.05617

[78] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating Accurate Assert Statements for Unit Test Cases using Pretrained Transformers. In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*. ACM/IEEE, 54–64. https://doi.org/10.1145/3524481.3527220

[79] Muhammad Usman, Wenxi Wang, Marko Vasic, Kaiyuan Wang, Haris Vikalo, and Sarfraz Khurshid. 2020. A study of the learnability of relational properties: model counting meets machine learning (MCML). In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1098–1111. https://doi.org/10.1145/3385412.3386015

[80] Julian von der Mosel, Alexander Trautsch, and Steffen Herbold. 2023. On the Validity of Pre-Trained Transformers for Natural Language Processing in the Software Engineering Domain. *IEEE Trans. Software Eng.* 49, 4 (2023), 1487–1507. https://doi.org/10.1109/TSE.2022.3178469

[81] Hailong Wang, Tongtong Xu, and Bei Wang. 2024. Deep Multiple Assertions Generation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering, FORGE 2024, Lisbon, Portugal, 14 April 2024*, David Lo, Xin Xia, Massimiliano Di Penta, and Xing Hu (Eds.). ACM, 1–11. https://doi.org/10.1145/3650105.3652293

[82] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Software Eng.* 50, 4 (2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208

[83] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.685

[84] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1398–1409. https://doi.org/10.1145/3377811.3380429

[85] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 191–200. https://doi.org/10.1145/1985793.1985820

[86] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1556–1560. https://doi.org/10.1145/3368089.3417943

[87] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[88] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S. Lee. 2023. Impact of Large Language Models on Generating Software Specifications. *CoRR* abs/2306.03324 (2023). https://doi.org/10.48550/ARXIV.2306.03324 arXiv:2306.03324

[89] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 557–569. https://doi.org/10.1145/3691620.3696020

[90] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. 2024. MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases. *ACM Trans. Softw. Eng. Methodol.* 33, 6 (2024), 150. https://doi.org/10.1145/3656340

[91] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1703–1726. https://doi.org/10.1145/3660783

[92] Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M. Atlee. 2023. Dynamic Human-in-the-Loop Assertion Generation. *IEEE Trans. Software Eng.* 49, 4 (2023), 2337–2351. https://doi.org/10.1109/TSE.2022.3217544

[93] Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2024. Generating Exceptional Behavior Tests with Reasoning Augmented Large Language Models. *CoRR* abs/2405.14619 (2024). https://doi.org/10.48550/ARXIV.2405.14619 arXiv:2405.14619

[94] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2024. Exploring Automated Assertion Generation via Large Language Models. *ACM Trans. Softw. Eng. Methodol.* (Oct. 2024). https://doi.org/10.1145/3699598 Just Accepted.

[95] Yifan Zhang, Dave Towey, and Matthew Pike. 2023. Automated Metamorphic-Relation Generation with ChatGPT: An Experience Report. In *47th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2023, Torino, Italy, June 26-30, 2023*, Hossain Shahriar, Yuuichi Teranishi, Alfredo Cuzzocrea, Moushumi Sharmin, Dave Towey, A. K. M. Jahangir Alam Majumder, Hiroki Kashiwazaki, Ji-Jiang Yang, Michiharu Takemoto, Nazmus Sakib, Ryohei Banno, and Sheikh Iqbal Ahamed (Eds.). IEEE, 1780–1785. https://doi.org/10.1109/COMPSAC57700.2023.00275

[96] Zhiyuan Zhong, Sinan Wang, Hailong Wang, Shaojin Wen, Hao Guan, Yida Tao, and Yepang Liu. 2024. Advancing Bug Detection in Fastjson2 with Large Language Models Driven Unit Test Generation. *CoRR* abs/2410.09414 (2024). https://doi.org/10.48550/ARXIV.2410.09414 arXiv:2410.09414