Do LLMs Generate Useful Test Oracles? An Empirical Study with an Unbiased Dataset

Davide Molinelli
Constructor Institute of Technology
Università della Svizzera italiana
Lugano, Switzerland

Luca Di Grazia
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Alberto Martin-Lopez
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Michael D. Ernst
Allen School of Computer Science & Engineering
University of Washington
Seattle, USA

Mauro Pezzè
Constructor Institute of Technology
Università della Svizzera italiana
Lugano, Switzerland

***Abstract*—Generation of thorough test oracles is an open problem. Popular test case generators, like EvoSuite and Randoop, rely on implicit, rule-based, and regression oracles that miss failures that depend on the semantics of the program under test. Formal specifications can yield test oracles but are expensive to create. Large Language Models (LLMs) have the potential to overcome these limitations. The few studies of using LLMs to generate test oracles use modest-sized public benchmarks, such as Defects4J, that are likely to be included in the LLM training data, which threatens the validity of the results. This paper presents an empirical study of the effectiveness of LLMs in generating test oracles. Our experiments use 13,866 test oracles, from 135 Java projects, that were created after the LLMs training cut-off dates. Thus, our dataset is unbiased. In our experiments, LLMs generated oracles with average mutation score of 43% — similar to the 45% score of human-designed test oracles. Our results also indicate that the test prefix and the methods called in the program under test provide sufficient information to generate good oracles, while additional code context does not bring relevant benefits. These findings provide actionable insights into using LLMs for automatic testing and highlight their current limitations in generating complex oracles.**

***Index Terms*—Software Testing, AI for Software Engineering, Oracle Generation**

I. Introduction

A test case consists of an input and an oracle [1]. An oracle is a predicate, such as an assertion, that determines whether the test case passed or failed [2]. Writing and maintaining test oracles is demanding and error-prone [3]. Manually writing test oracles for automatically generated test cases is infeasible, and test oracles are the main bottleneck for automatic test generation [4].

Popular test case generators, like EvoSuite [5] and Randoop [6], rely on weak implicit oracles and on regression oracles that suffer from false positives. One promising approach to automatically generating test oracles is via natural language processing (NLP), starting from natural language documentation of the code under test [7]–[9]. However, these NLP-based approaches do not generalize to poorly commented code [7].

Large Language Models (LLMs) are trained on massive corpora of both code and documentation [10], [11]. LLMs can synthesize semantically meaningful assertions by generalizing from context, even for poorly commented code [12]. LLMs can generate both assertion oracles and test oracles [13], [14]. *Assertion oracles*, or *axiomatic oracles* [15], are valid for all inputs [16]–[20]. They may be written in the unit under test or in the test case [21]. *Test oracles*, or *concrete oracles* [21], [22], are assertions in the test case that are specific to that test input [23]–[25].

Previous work evaluates LLMs on modest-sized public datasets [17], such as Defects4J [26]. This well-known benchmark is likely to have been included in LLM training data; such data leakage may inflate performance estimates [27]–[29]. A recent survey [30] confirms the absence of large-scale evaluation to assess the effectiveness of LLMs in generating assertions for test cases added after the LLM's training cut-off. This highlights a critical gap in understanding the true generalization ability of LLMs for generating oracles [31].

A. Methodology

This paper fills the gap with a large-scale empirical study of concrete test oracles generated by LLMs on a dataset designed to avoid leakage from the training set. We extracted 13,866 oracles from 135 open-source Java projects. All of these test cases were created after 2024-09-01, ensuring that the test code was not in the models' training data. We evaluated 10 LLMs from 3 families (llama, phi, and qwen), including general-purpose, code-specific, and reasoning-enhanced variants, with 4 different prompt configurations. Our experiments generated 610,104 oracles: 13,866 oracles per LLM-and-prompt pair.

We evaluated the LLM-generated oracles in two ways: by comparing them to the original programmer-written oracles (a proxy for correctness) and by computing their contribution to mutation score (a proxy for error detection).

B. Results

Our experiments demonstrate that LLMs can generate a large number of useful test cases. Figure 8 indicates that

LLM-generated and programmer-written oracles have similar mutation scores: 43% vs. 45%.

As in previous experiments, larger LLMs perform better. However, in our study the training specialization and the model family do not matter. In contradiction to the common expectation, for this coding task general-purpose LLMs performed as well as code-focused models.

The most surprising result of the empirical study is that adding code context information about the focal and test classes does not improve the performance over prompting the LLM with the test prefix and the called methods only. The empirical study also indicates that the LLMs perform similarly for different types of oracles.

C. Contributions

This paper makes the following contributions:

- A benchmark of 13,866 oracles and a methodology to build new ones. These test cases postdate the LLM training cut-off dates, enabling unbiased evaluation of test oracle generation.
- An empirical evaluation of 10 LLMs from 3 families (llama, phi, and qwen) across different LLM sizes, prompt strategies, and assertion types (that is, which `assert*` method performs the check), including an automated mutation analysis.
- Insights for researchers and practitioners into the effectiveness and limitations of LLM-generated assertions.

The paper is organized as follows. Section II summarizes our implementation, and section III explains methodology: how we collected the dataset and ran the mutation analysis. Section IV presents the experimental results. Section V discusses our findings and implications for researchers and practitioners. Section VI underlines the limitations and threats to validity. Section VII overviews related work.

II. Implementation and data availability

We implemented our methodology in Java and Python, using open-source libraries (like `JavaParser` [32], `tree-sitter` [33], and `PyDriller` [34]), interfacing with the Ollama [35] and vLLM [36] libraries. The infrastructure is LLM-independent, so the study can be replicated using different LLMs.

We provide a replication package to reproduce our results: https://github.com/darthdaver/llm-prompts-empirical-study.

III. Methodology

Figure 1 shows an overview of our methodology, which is centered around a dataset of 13,866 *Test Oracles* that we mined from GitHub repositories (Section III-A) and that we query with *4 Prompts* (section III-B) to evaluate a set of *LLMs* (section III-C) with metrics and mutation analysis (section III-D). Figure 2 summarizes the numbers and configurations of our empirical study.

A. Dataset

We built an unbiased dataset of test cases by selecting candidate repositories, then extracting recent test cases.

Algorithm 1 Extracting test cases created after a given date.

```
Input: repository $r$, starting date $since$
Output: test cases $T_{new}$ created after $since$
$commits \leftarrow$ PyDriller$(r, since)$    ▷ date-ordered
$T_{new} \leftarrow \{\}$    ▷ recent tests; the output of this procedure
for $c \in commits$ (main branch) do
    for $f \in c.modified\_files$ do
        if IsJavaTestClass$(f)$ then
            $code_{before} \leftarrow source\_code\_before(c, f)$
            $code_{after} \leftarrow source\_code\_after(c, f)$
            $T_{before} \leftarrow$ GetTestMethods$(code_{before})$
            $T_{after} \leftarrow$ GetTestMethods$(code_{after})$
            $T_{add} \leftarrow T_{after} \setminus T_{before}$
            $T_{del} \leftarrow T_{before} \setminus T_{after}$
            $T_{mod} \leftarrow$ ExtractTestDiff$(T_{after}, T_{before})$
                    ▷ $T_{add}$, $T_{mod}$, and $T_{del}$ are disjoint
            $T_{new} \leftarrow (T_{new} \cup T_{add} \cup T_{mod}) \setminus T_{del}$
return $T_{new}$

procedure GetTestMethods$(code)$
    if $code = None$ then return $\emptyset$
    $tree \leftarrow$ TreeSitterParse$(code)$
    return $\{m \in tree\ .\ m$ is a test method$\}$

procedure ExtractTestDiff$(T_{after}, T_{before})$
        ▷ Extracts the test cases added after the since date
    and whose body has been modified in the last commit.
    $T_{mod} \leftarrow \emptyset$
    for all $t_a \in T_{after}$ do
        if $t_a$ added after $since$ then
            $t_b \leftarrow$ find$(T_{before}, (t) \rightarrow \{t.sig = t_a.sig\})$
            if $t_b \neq$ None then
                if $t_a$.body $\neq t_b$.body then
                    $T_{mod} \leftarrow T_{mod} \cup \{t_a\}$
    return $T_{mod}$
```

1) Selecting candidate repositories: We selected public, non-fork repositories from GitHub, using SEART GitHub Search [37] with the following filters:

(i) at least **100** commits,
(ii) at least **50** issues,
(iii) at least **10** distinct contributors,
(iv) at least **10** stars,
(v) at least 1 commit between **2024-09-01** and **2025-05-05**.

This yielded 4,517 candidate repositories. To simplify the mutation analysis of section III-D, we retained the repositories with a single `pom.xml` file that successfully compile with Maven. This resulted in 135 Maven projects. We did not check whether the test cases pass. Later, we performed mutation analysis only on a suite of passing test cases.

2) Extracting recent test cases: We built an unbiased repository of programmer-written test cases by extracting the test cases introduced in the candidate projects after 2024-09-01, according to the commit history of the projects.

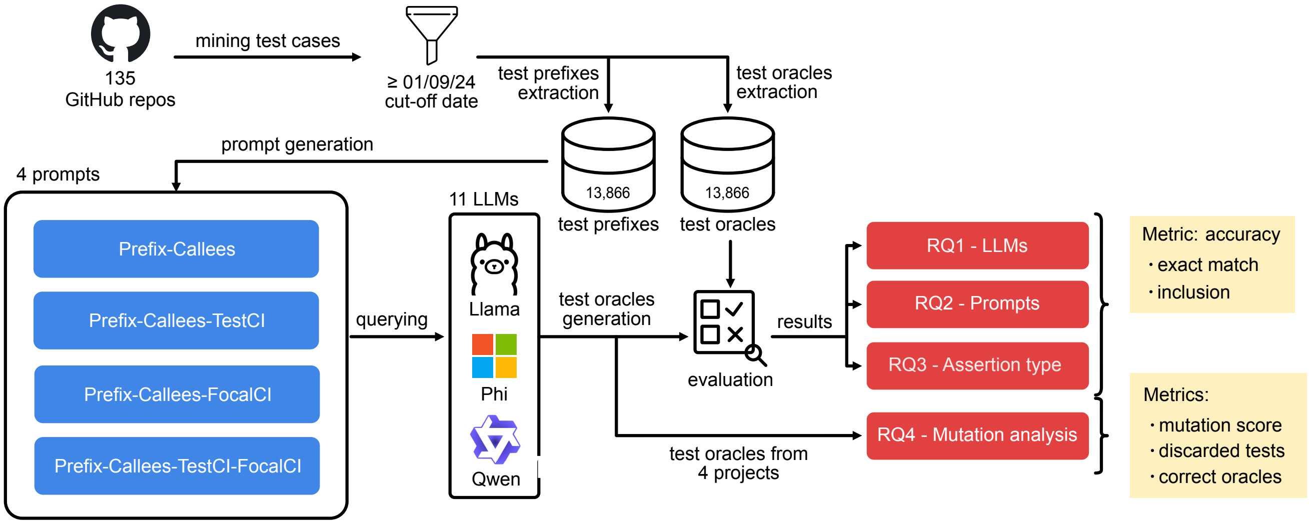


Fig. 1: Overview of our methodology.

Fig. 2: Corpus and configuration of our empirical study.

Candidate repositories	4,517
Repositories that compile with Maven	135
Manually designed test oracles	13,866
Time window of mined commits	2024-09-01 to 2025-05-05
Evaluated LLMs	10
Prompt strategies per oracle	4
Concrete prompts (#oracles × #strategies)	55,464
Total LLM queries (#concrete prompts × #LLMs)	610,104

A test oracle is an `assert` statement in the test case. The *test prefix* of an oracle is the code in the test case up to the given oracle, including any prior oracles. We obtained a set of 13,866 test oracles with their prefixes, manually added by the project developers after 2024-09-01.

Algorithm 1 outlines the procedure to find every test that was created after the start date, and then for each of those tests we retrieve its final version. For each commit after a given date in chronological order (lines 3–14), it identifies the test files that have been modified (line 4), analyzes the source code of the new and old versions of the test file to extract the test cases defined in both versions (if the file is added in the current commit, the set of test cases in the old version is empty) in lines 6–7 , adds the new test cases to the target set T_{new}, removes from T_{new} any test case added in a previous commit and removed in the current one, and updates the body of any test case added in a previous commit and modified in the current one (line 14) based on fully identical definitions. It returns the test cases collected (line 15).

Algorithm 2 shows how we build a *Dataset* of oracles (`assert` statements) with the corresponding test prefix, the focal class, and the test class, from the test cases that algorithm 1 extracts from the repositories.

The *prefix* contains the statements of the test case that precede the target assertion. The prefix also contains the assertions that preceded the target assertion, if any. The *focal class* is the class that contains the methods that the test case tests. A test case might call other methods (including methods from other classes) as well, in the prefix, to build and prepare objects or set up a state for the call to the method under test. Our dataset contains unit test cases. In a unit test, the method under test (the focal method) is typically called immediately before the assert statement, but it also might be called earlier or called in the assertion (see fig. 3). Our methodology does not attempt to determine what the focal method is. Instead, it takes advantage of the standard unit test convention that test class `MyClassTest` tests the methods of source code class `MyClass` (see algorithm 2, line 4).

State-of-the-art approaches use the heuristic that the last method call in the test case is the focal method [17], [23], [38]. This is a strong assumption that does not always hold. For example, the test case `test12()` in fig. 3 tests the method `shift` called before the `assertEquals` statement. The algorithm extracts the definition of the last method called in the original test case (`length`) and considers it as the focal method to train the model to predict the target assertion. However, the method `length` only represents a getter used to check the post-condition of the method `shift`. Our methodology does not treat any method call in the test case differently than others; rather, it collects the definitions of all the constructors and methods invoked within the test prefix.

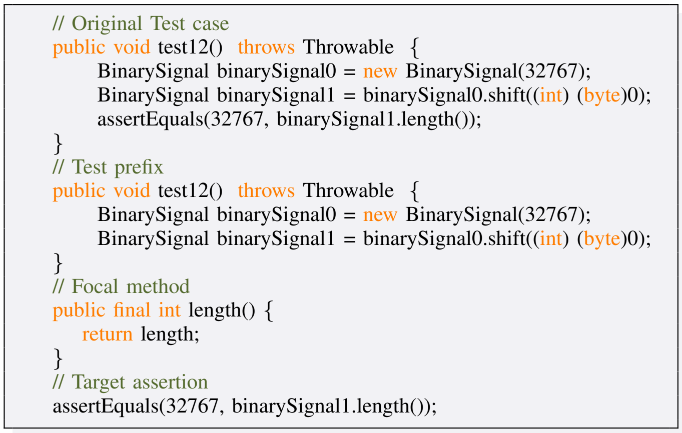
A test case can contribute more than one oracle if it contains more than one assertion. In this case, the number of datapoints obtained by processing the original test case is equal to $|A|$.

The final dataset contains test cases with 1 (34.4%), 2 (16.6%), 3 (11.5%), and 4 or more (37.5%) assertions. In the case of multiple assertions in the same test case, the prefix of each assertion includes the assertions that occur above in the test case. The many test cases with single assertions in the dataset make our study applicable for generating assertions for

Algorithm 2 Building dataset D of oracles.

```
Input: repository $r$, test cases $T$ from algorithm 1
Output: dataset $D$ of oracles, where each element is a 5-tuple $\langle tc, prefix, fc, invoked, a\rangle$, with test class $tc$, test prefix $prefix$, focal class $fc$, $invoked$ is the methods called in prefix, and $a$ is the next assertion to predict
$D \leftarrow \emptyset$    ▷ dataset of oracles; the output of this function
$S \leftarrow \emptyset$    ▷ set of ⟨focal class, test class⟩ pairs
for $fc$ in SOURCECLASSES($r$) do
    $tc\_name \leftarrow fc.name +$ 'Test'    ▷ test method name
    $tc \leftarrow$ GETTESTCASE($tc\_name$)
    if $tc \neq$ None then
        $S \leftarrow S \cup \langle fc, tc\rangle$
for $\langle fc, tc\rangle \in S$ do
    for $t \in tc.tests$ do
        $t.body \leftarrow$ ENRICHTESTCASE($t.body$)
        $A \leftarrow$ EXTRACTASSERTIONS($t.body$)
        for $a \in |A|$ do
            $prefix \leftarrow t.body.stmts[0..a]$    ▷ all statements (and previous assertions if any), before $a$
            for method call $mc$ in $t.body$ do
                $invoked \leftarrow mc.decl \cup mc.decl.body$
            $D \leftarrow D \cup \{\langle tc, prefix, fc, invoked, a\rangle\}$
return $D$
procedure ENRICHTESTCASE($body$)    ▷ Embed bodies of procedures containing assertions and called within the test case
    for method call $mc$ in $body$ do
        if ISMETHODWITHASSERTIONS($mc$) then
            /* The statements and the assertions of
            the method are integrated within the test case.
            The variables are refactored to guarantee
            consistency. */
            EMBEDBODY($body, mc.decl.body$)
    return $body$
```

Fig. 3: A test case with one assertion, from the TOGLL replication package [17].



```
// Original Test case
public void test12()  throws Throwable  {
      BinarySignal binarySignal0 = new BinarySignal(32767);
      BinarySignal binarySignal1 = binarySignal0.shift((int) (byte)0);
      assertEquals(32767, binarySignal1.length());
}
// Test prefix
public void test12()  throws Throwable  {
      BinarySignal binarySignal0 = new BinarySignal(32767);
      BinarySignal binarySignal1 = binarySignal0.shift((int) (byte)0);
}
// Focal method
public final int length() {
    return length;
}
// Target assertion
assertEquals(32767, binarySignal1.length());
```

Algorithm 3 Building prompts.

```
Input: oracle dataset $D$ from algorithm 2, and a prompt specifier $pspec$ that contains two Boolean fields $includeFocalClass$ and $includeTestClass$
Output: one prompt for each oracle in the dataset, as specified in the prompt type
procedure BUILDPROMPT($pspec$)
    $prompts \leftarrow \{\}$
    for $\langle tc, prefix, fc, \_, \_\rangle \in D$ do
        $p \leftarrow d.prefix$
        $ims \leftarrow tp.invoked$
        for $m \in ims$ do
            $p \leftarrow p \,\|\, m$
        if $pspec.includeFocalClass$ then
            ADDCLASSINFO($p, fc$)
        if $pspec.includeTestClass$ then
            ADDCLASSINFO($p, tc$)
        $prompts \leftarrow prompts \cup \{p\}$
    return $prompts$

procedure ADDCLASSINFO($p, c$)
    for $f \in c.fields$ do
        $p \leftarrow p \,\|\, f.signature$
    for $m \in c.methods$ do
        $p \leftarrow p \,\|\, m$
```

test prefixes without assertions, for instance, for automatically generated test cases, like EvoSuite test cases without regression oracles. The many test cases with multiple assertions in the dataset make our study applicable also for adding assertions to test cases that already include some assertions, like manually generated test cases.

B. Prompts

We defined 4 prompts with different information content [39]:

Prefix-Callees The prompt contains the prefix of the oracle and the source code of the methods called in the test case.

Prefix-Callees-TestCl The prompt contains the prefix of the test case, the source code of the methods called in the test case, and the definitions of fields and methods (Javadoc, signature, and body) in the test class.

Prefix-Callees-FocalCl The prompt contains the prefix of the test case, the source code of the methods called in the test case, and the definitions of fields and methods (Javadoc, signature, and body) in the focal class.

Prefix-Callees-TestCl-FocalCl The prompt contains the prefix of the test case, the source code of the methods called in the test case, and the definitions of fields and methods (Javadoc, signature, and body) in both the focal and the test class.

Algorithm 3 outlines how we automatically build the prompts. The algorithm generates the prompts according to a prompt template for all the oracles in an input dataset. We use the algorithm to generate the 55,464 prompts for our experiments (4 prompts for each of the 13,866 oracles).

The algorithm extracts the elements required to build the prompt (prefix, signature, Javadoc and body of the methods invoked in the test prefix, and fields and methods defined within the focal class and the test class [40], depending on the prompt type) and initializes an empty list *prompts* of prompts (lines 1–2). The algorithm iterates over all the oracles in the input dataset D, and generates the prompt specified in the input template for each of them (lines 3–12) that it adds to the output set of *prompts* (line 12). In each iteration, the algorithm initializes the prompt with the test prefix and the information about the methods invoked within it (lines 5–7). It adds the information required for the focal (lines 8–9) and test class (lines 10–11) with procedure ADDCLASSINFO if indicated in the prompt type. The configuration file contains the types of information to be included to progressively compose the prompt with the signature, the Javadoc and the body of the methods invoked in the test prefix, and eventually the fields and the methods defined within the focal class and the test class (line 1). The procedure iterates over the **13,866** oracles data points collected in the previous phase, integrating the pieces of information within the prompt (lines 3–12). The algorithm enriches the prompt with a clear description of the task that each model must accomplish and generates as output a list of **13,866** final prompts (line 13).

We fit as much information as possible in the prompts while respecting the maximum window context length of the model, by leveraging an algorithm that implements the E-Wash approach [41].

We built the 4 prompts for each of the 13,866 oracles in our dataset, for a total of 55,464 prompts that we used to query the LLMs to generate a test oracle without any example (*zero-shot* learning).

C. Large Language Models

To study how *model size* and *training specialization* impact oracle prediction, we selected 10 different LLMs that use different training and inference techniques: **General**, **Reasoning**, **Instruct**, and **Code**, and size ranging from **1B** to **70B** parameters.

General. General-purpose LLMs are trained predominantly on web-scale mixed-domain datasets:

- qwen2.5_1.5b, qwen2.5_14b, qwen2.5_32b
- phi4-mini_3.8b, phi4_14b

Reasoning. LLMs with pretraining and instruction-tuning that explicitly target chain-of-thought and analytical reasoning tasks.

- phi4-reasoning_14b

Instruct. LLMs fine-tuned with human-based feedback or datasets specialized in natural-language instructions.

- llama3.3_70b

Algorithm 4 Mutation analysis for generated oracles.

```
Input: repository $r$, oracle dataset $D$ from algorithm 2
Output: Mutation score with and without the LLM-generated
    assertions
$green\_suite \leftarrow \epsilon$
for $\langle tc, \_, \_, \_, \_ \rangle \in D$ do
    $green\_suite \leftarrow$ DUPLICATE($tc$)
    $green\_suite.body \leftarrow \epsilon$
    for $t \in tc.tests$ do
        $green\_suite.body \leftarrow green\_suite.body \cup t$
        if PASSINGTESTS($r$,$tc$) == False then
            $green\_suite.body \leftarrow green\_suite.body \setminus t$
    $green\_suite \leftarrow green\_suite \cup green\_suite$
$score_{orig} \leftarrow$ RUNPIT($r$, $green\_suite$)
for $tc \in green\_suite$ do
    for $t \in tc.tests$ do
        REMOVEORIGINALASSERTION($t$)
$score_{without} \leftarrow$ RUNPIT($r$, $green\_suite$)
for $tc \in green\_suite$ do
    for $t \in tc.tests$ do
        ADDGENERATEDASSERTION($t$)
        if PASSINGTESTS($r$,$tc$) == False then
            REMOVEGENERATEDASSERTION($t$)
$score_{gen} \leftarrow$ RUNPIT($r$, $green\_suite$)
for $tc \in green\_suite$ do
    for $t \in tc.tests$ do
        ADDORIGINALASSERTION($t$)
        if PASSINGTESTS($r$,$tc$) == False then
            REMOVEORIGINALASSERTION($t$)
$score_{orig+gen} \leftarrow$ RUNPIT($r$, $green\_suite$)
return $\langle score_{orig}, score_{without}, score_{gen}, score_{orig+gen} \rangle$
```

Code. LLMs trained or fine-tuned on source code datasets; expected to excel at syntactic and semantic understanding typical of coding tasks.

- qwen2.5-coder_1.5b, qwen2.5-coder_14b, qwen2.5-coder_32b

We selected LLMs of the same family with multiple sizes (e.g. qwen2.5 at 1.5B, 14B, and 32B) to understand the effect of scale from that of training data. Moreover, by comparing domain-specialized LLMs (Code and Reasoning) with their general versions, we evaluate whether the oracle generation tasks benefit more from scale or from task-aligned pre-training.

D. Mutation Analysis

We compute the mutation score across four different scenarios to evaluate the impact of the generated oracles. We measure: (i) the mutation score of the original test cases (including the test oracles written by the developers); (ii) the mutation score of the test prefixes, by executing the test cases stripped of all `assert` statements; (iii) the mutation score with only the generated oracles, by executing the test cases augmented with the LLM-generated oracles; and (iv) the mutation score with both original and generated oracles combined. These four configurations allow us to isolate the effect of generated oracles and assess their contribution both independently and in conjunction with existing ones.

Algorithm 4 outlines the process to compute the mutation scores for the classes of a repository. The algorithm initializes the suite of green test classes to an empty set (line 1) and iterates over the list of test classes in the dataset D of oracle datapoints to verify that their tests compile and pass, otherwise it discard them (lines 2–8). The algorithm adds the refined test classes to the suite (line 9), and computes the mutation score of the tests containing the original assertions (lines 2–9). The algorithm removes the original test oracles written by the developers from the test cases of the test classes in the suite (lines 11–13) and computes the mutation score of the tests without oracles (line 14). The algorithm adds the LLM-generated oracles to the test cases without oracles, only if the updated test cases compile and pass, (lines 15–19) and computes the mutation score of the test cases containing only the generated oracles (line 20). Finally, the algorithm includes the original oracles to the test cases containing the LLM-generated oracles (lines 21–25) and computes the mutation score of the test cases embedding both the original and the generated oracles (line 26).

IV. Results

In this section we describe our experimental setup and the results of our empirical study.

A. Research Questions

Our experimental evaluation addresses the following research questions (RQs):

RQ1 *Impact of the LLM: Does the choice of LLM affect the accuracy of the generated oracles?* We compare different LLMs to evaluate the influence of the type and size of the LLM on the generation process.

RQ2 *Impact of the Prompt: Does the choice of prompt affect the accuracy of the generated oracles?* We evaluate different input information to study the tradeoff between the amount of information provided with the prompt and the effectiveness of the LLM.

RQ3 *Impact of the Type of Oracle: Are some assertion methods easier for LLMs to accurately generate?* We compare the accuracy of the LLM output for different `assert*` methods.

RQ4 *Effectiveness of the Oracles: Do the generated oracles improve the mutant detection ability of the test cases?* We compare the mutation score of the test cases with and without the generated oracles.

Figure 4 does not report the time for llama3.3 because it was run on a different cluster and the measurements are incompatible.

B. Experimental Setting

To generate an oracle, our experiments remove the programmer-written oracle from the test case, prompt an LLM to generate an oracle for the test prefix, and compare the LLM-generated oracle with the original programmer-written one. Thus, we use the 13,866 oracles manually designed by the 135 projects' developers as baseline.

Overall, our experiments prompted 10 LLMs with 4 different prompts for each oracle, for a total of 610,104 queries.

For RQ1–RQ3, we measure the accuracy of LLM-generated oracles in terms of their similarity to the original programmer-written oracles. An exact match is when two oracles are character-for-character identical. An inclusion match is when one of the two oracles is a substring of the other. We refined the limited automatic comparison by running the mutation analysis on a sample set of oracles to measure the effectiveness of the oracles generated (RQ4). We executed all the experiments on an Ubuntu 20.04 cluster with four NVIDIA A100 GPUs, each with 40 GB of VRAM.

String matching and substrings can be easily automated, which is essential for a large-scale experiment such as ours. However, both measures are pessimistic under-approximations of the accuracy of the LLMs. Exact matching misses generated oracles that semantically match the original oracles, while expressed in a different syntactic form. We observed that exact matches sometimes failed due to small and easily fixable syntactic differences, like a missing semicolon.

C. RQ1 Impact of the LLM on the Generated Oracles

We measure the effectiveness of the LLM (research question RQ1) as the percentage of oracles that the LLM generates with respect to the original oracles.

Figure 4 reports the accuracy of the set of generated oracles for each combination of LLMs and prompts. Column "no oracle" indicates the percentage of outputs of the LLM that are not oracles, that is, they do not contain an assert statement. In any row, the "exact match" column is a subset of the "inclusion" column, and the "inclusion" column plus the "no oracle" column is less than or equal to 100%.

The best-performing models are the largest models: qwen2.5-32b, qwen2.5-coder-32b, llama3.3-70b, qwen2.5-coder-14b, qwen2.5-14b, phi4-14b. The only exception is the reasoning model phi4-reasoning-14b that performs particularly poorly despite its size. It usually generates no oracle (over 75% of the time as indicated in column "no oracles").

Figure 4 indicates similar performance for different types of model. For example, qwen2.5-coder behaves similarly to qwen2.5 at each model size. It is surprising that the code models perform similarly to the general models at our code generation task. We hypothesize that the best performance of qwen-2.5 is due to a training set particularly rich in code, as stated in the Qwen2.5's technical report: *The pre-training data increased from 7 trillion tokens to 18 trillion tokens, with focus on knowledge, coding, and mathematics* [42].

RQ1 Findings: The size of the LLM matters, while the type and model do not. Scaling from 1.5b to 32b parameters multiplies *inclusion* accuracy almost ten-fold (3% to 29%), and it is consistent across both code and generic LLMs.

Fig. 4: Accuracy for each combination of LLMs and prompts. The models are sorted by the "average" column.

LLM			**Prompt**	**Accuracy**						**Time**
Type	**Model**	**Size**	**Type**	**exact match**		**inclusion**				**(s)**
				#	**%**	**#**	**%**	**avg.**	**no oracle**	
Code	qwen2.5-coder	32b	Prefix-Callees	806	5.8%	3,993	28.8%	29.4%	8.4%	1.04
			Prefix-Callees-TestCl	673	4.8%	4,067	29.3%		8.0%	4.05
			Prefix-Callees-FocalCl	821	5.9%	4,010	28.9%		7.2%	1.16
			Prefix-Callees-TestCl-FocalCl	621	4.5%	**4,266**	**30.7%**		7.4%	4.17
General	qwen2.5	32b	Prefix-Callees	2,923	21.0%	3,960	28.5%	29.2%	5.0%	1.06
			Prefix-Callees-TestCl	2,883	20.8%	4,100	29.5%		4.2%	4.08
			Prefix-Callees-FocalCl	2,944	21.2%	4,026	29.0%		5.0%	1.18
			Prefix-Callees-TestCl-FocalCl	**3,762**	**27.1%**	4,124	29.7%		3.9%	4.28
Instruct	llama3.3	70b	Prefix-Callees	1,379	9.7%	3,118	21.9%	22.9%	3.9%	N/A
			Prefix-Callees-TestCl	1,714	10.3%	3,868	23.2%		1.4%	N/A
			Prefix-Callees-FocalCl	1,463	10.5%	3,146	22.7%		1.5%	N/A
			Prefix-Callees-TestCl-FocalCl	1,911	11.4%	4,015	24.0%		1.2%	N/A
Code	qwen2.5-coder	14b	Prefix-Callees	968	7.0%	2,946	21.2%	20.9%	1.6%	0.58
			Prefix-Callees-TestCl	889	6.4%	2,790	20.1%		3.4%	2.18
			Prefix-Callees-FocalCl	939	6.8%	2,976	21.4%		2.0%	0.64
			Prefix-Callees-TestCl-FocalCl	943	6.8%	2,870	20.7%		3.0%	2.28
General	qwen2.5	14b	Prefix-Callees	233	1.7%	2,910	21.0%	20.7%	1.8%	0.59
			Prefix-Callees-TestCl	337	2.4%	2,866	20.6%		3.2%	2.18
			Prefix-Callees-FocalCl	362	2.6%	2,886	20.8%		1.7%	0.66
			Prefix-Callees-TestCl-FocalCl	316	2.3%	2,823	20.3%		2.5%	2.29
General	phi4	14b	Prefix-Callees	1,562	11.2%	2,566	18.5%	17.8%	0.8%	0.78
			Prefix-Callees-TestCl	1,429	10.3%	2,424	17.5%		3.8%	2.69
			Prefix-Callees-FocalCl	1,515	10.9%	2,464	17.7%		1.0%	0.91
			Prefix-Callees-TestCl-FocalCl	1,426	10.3%	2,398	17.3%		3.7%	3.36
General	phi4-mini	3.8b	Prefix-Callees	560	0.1%	642	4.6%	4.6%	0.9%	0.60
			Prefix-Callees-TestCl	13	0.1%	634	4.6%		2.9%	1.42
			Prefix-Callees-FocalCl	28	0.2%	605	4.4%		0.9%	0.76
			Prefix-Callees-TestCl-FocalCl	14	0.1%	636	4.6%		2.1%	1.44
Reasoning	phi4-reasoning	14b	Prefix-Callees	16	0.1%	642	4.6%	4.6%	77.3%	38.47
			Prefix-Callees-TestCl	13	0.1%	634	4.6%		76.3%	42.11
			Prefix-Callees-FocalCl	28	0.2%	605	4.4%		78.3%	36.17
			Prefix-Callees-TestCl-FocalCl	14	0.1%	636	4.6%		75.3%	41.99
General	qwen2.5	1.5b	Prefix-Callees	148	1.1%	536	3.9%	3.3%	3.6%	0.24
			Prefix-Callees-TestCl	39	0.3%	403	2.9%		7.2%	0.61
			Prefix-Callees-FocalCl	90	0.6%	486	3.5%		5.4%	0.27
			Prefix-Callees-TestCl-FocalCl	54	0.4%	395	2.8%		6.5%	0.62
Code	qwen2.5-coder	1.5b	Prefix-Callees	155	1.1%	436	3.1%	2.7%	12.6%	0.21
			Prefix-Callees-TestCl	120	0.9%	339	2.4%		17.6%	0.56
			Prefix-Callees-FocalCl	147	1.1%	410	3.0%		14.8%	0.22
			Prefix-Callees-TestCl-FocalCl	126	0.9%	329	2.4%		15.3%	0.58

D. RQ2 Impact of the Prompt on the Generated Oracles

We measure the effectiveness of the information used to prompt the LLM as the percentage of oracles that the LLM generates with different information in the prompt. Surprisingly, the difference among the 4 is irrelevant for any of the models, both for exact match and inclusion. Thus, adding information (focal and test class) to the prefix and call increases the size of the prompt without significantly improving the oracle generation. The results indicate that the code, being it the focal class or the test class, does not include relevant information about the oracles.

RQ2 Findings: A prompt containing the test prefix and called methods provides enough information to generate oracles. Additional information like focal class and test class does not significantly improve the generated oracles.

E. RQ3 Difficulty of Different Types of Oracle

Figure 5 shows the percentage of generated oracles that match the programmer-written ones, grouped by type of oracles (the `assert*` methods that the oracle calls). In the figure, we refer to *inclusion matching* to compare the original with the generated oracles. The category *Other* groups the types that contain fewer than 10 oracles.

The diagram reports the oracle type on the x-axis, and labels the bars with the percentage and the number of oracles of the given type. The bars are labeled with the total number of oracles and the percentage of matching oracles for each type.

The different distribution among the assertion types for nine of the types varies from 23.4% (`assertInstanceOf`) to 34% (`assertArrayEquals`), with a percentage over 35% for `assertNotEquals` (40.6%), and a percentage below 20% for four types of oracles. We observe that the type of oracle has some impact on the generation process, with some outliers performing both better and worse than the majority of types. However, the figure refers to the inclusion matching, which does not capture semantic matches that differ syntactically, and this does not allow us to generalize the result. Moreover, fig. 6 shows a summary where each datapoint is one of the 13,866 predicted by each LLM for each prompt. So, it contains all the 610,104 predictions of this study. The results are similar to fig. 5, showing also that there is not a specific configuration that works better on a specific type of assertion. Finally, we computed the Pearson correlation between the frequency of each assertion type and the matching rate: $r = 0.112, p = 0.72$, which underlines the non-correlation between these two factors.

RQ3 Findings: The effectiveness of the generation process depends on the type of oracles, albeit with no clear winners or losers; however, the approximation of *inclusion matching* does not allow us to generalize the result.

F. RQ4 Effectiveness of the Generated Oracles on the Test Cases

We measure the effectiveness of the generated oracles (research question RQ4) by comparing the mutation score of the test suite with generated oracles to the mutation score of the test suites with no oracles and with the original oracles. The comparison to no-oracles measures the impact of the generated oracles on the precision of the test suite. The comparison to the original oracles measures the effectiveness of the generation process with respect to well-designed oracles. The comparison of the mutation score of all oracles to the mutation score of the original oracles only measures the impact of the generated oracles on developers' test suites. We experimented with qwen2.5 32b prompted with text prefix, method call, test class, and focal class, one of the best performing configurations, executed on a sample set of projects that meet the requirements discussed in section III-D (projects that compile, execute and support PIT).

To compute the mutation score, we had to discard all oracles that fail since PIT requires a green test suite. Figure 7 reports the number of generated oracles that do not compile (*Compilation Failure*), fail (*Test Failure*), and pass (*Test Passes*) for the four projects we used in this research question. We discarded between 1 in *twilio/twilio-java* (17%), and 33 in *wmixvideo/nfe* (55%) oracles, 10 oracles on average (40%). We manually verified that the 8 oracles that fail (*Test Failure*) are incorrect (false positives). The test cases that fail and that PIT ignores may still be useful to uncover real bugs (for instance, failures due to correct oracles in a faulty program). We manually verified that the 39 oracles that pass (*Test Passed*) are correct (no false negatives), highlighting the potential of LLMs for creating oracles useful for mutation testing.

Figure 8 presents the mutation score we obtained for four projects without oracles (Column "Without oracles"), with the generated oracles (Column "With generated oracles") with the original oracles (Column "With original oracles") and with both original and generated oracles (column *Original + Generated oracles*). Figure 8 reports the mutation score in percentage. The results indicate that the generated oracles significantly improve the test cases without oracles, with a relative increment that ranges from 76% to 443%, and perform similarly to the original oracles, with a loss of 40% in the worst case and a gain of 30% in the best case, with an average loss of 10%. The generated oracles added to the original oracles improve the mutation score only for a project (`wmixvideo/nfe`), and gain the same score for the other three projects. While the improvement over the test cases with no oracles is largely expected, the behavior of being not too far and in one case even better than the original and very well-written oracles is quite surprising, and indicates the high quality of the generated oracles. The significant improvement (from 61% to 80%) of the generated oracles added to the original oracles in the only project where the generated oracles obtain a high mutation score indicates that a good number of generated oracles can improve over the original oracles.

We performed a qualitative analysis of all 79 LLM-generated oracles of the four projects and we observed that:

- Developers generate assertions with specific literals hardly predictable by LLMs (for instance, `%3C%3Fxml+version%3D%221.0%22+encoding%3D%22UTF-8%22%3F%3E%3CConversationRelay%2F%3E`). LLMs produce weaker still-correct assertions, usually containing references to previous elements of the test prefix (for instance, *elem.getElementAttributes().size()*).
- LLMs can extract patterns from test prefixes that contain multiple assertions, and correctly predict subsequent assertions, although LLMs seldom fail to understand the underlying semantics, sometimes due to missing context in the prompt.
- The two limitations above are rooted in the information retrieval techniques, and can be addressed with advanced information retrieval techniques that intelligently complement prompts with relevant contextual information.

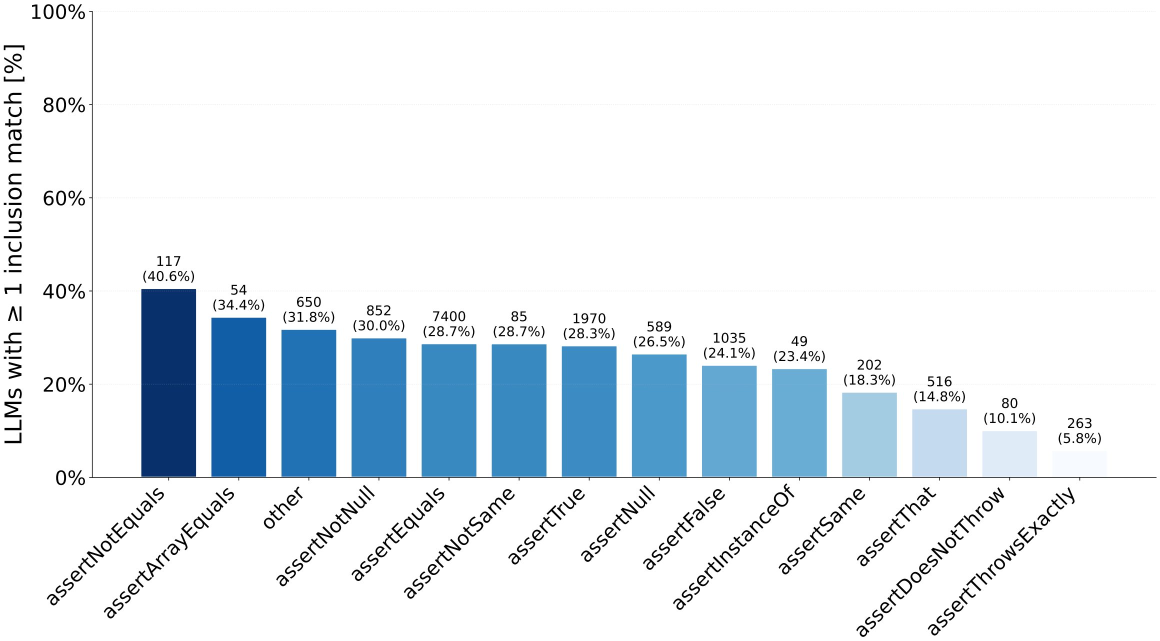


Fig. 5: For each assertion type or method, how often at least one of the four prompts of each LLM exactly matched the human-written oracle.

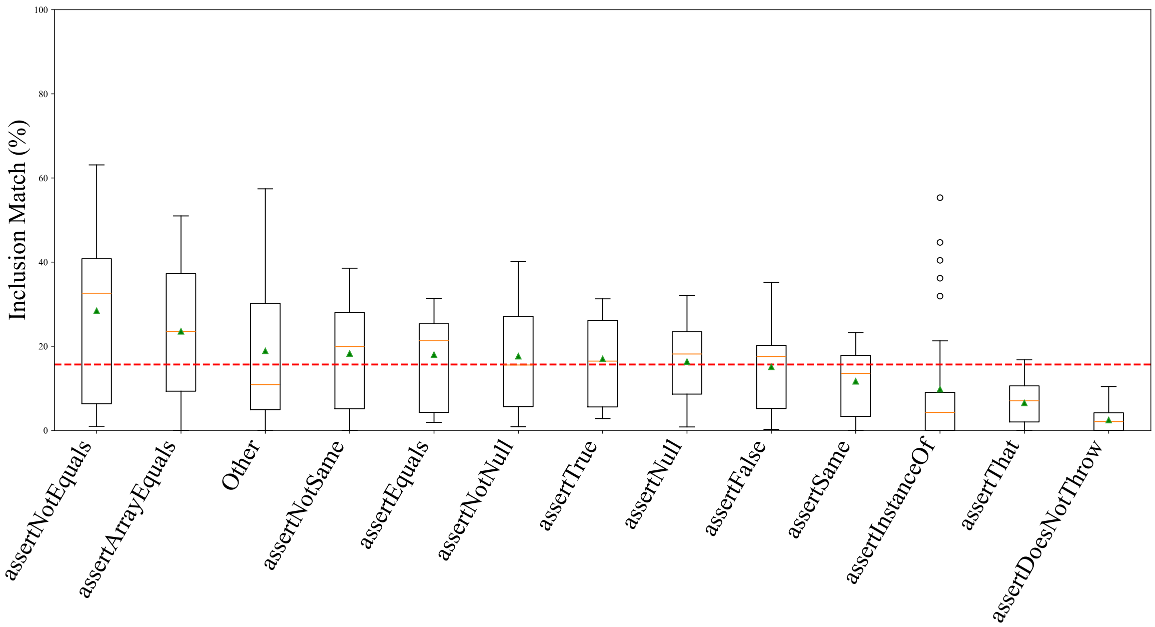


Fig. 6: Average of all the 610,104 oracle predictions per type of assertion (the dash line represents the overall mean).

Fig. 7: Number of test cases that do not compile, fail, and pass with the generated oracles.

Project	Compilation Failure	Test Failure	Test Passed
twilio/twilio-java	1	0	5
bkiers/liqp	0	2	3
wmixvideo/nfe	29	4	27
wearefrank/ladybug	2	2	4
TOTAL	32	8	39

Fig. 8: Mutation scores (%) with different test oracles. The test code is identical, and only the oracles differ.

Project	**Original oracles**	**Without oracles**	**Generated oracles**	**Original + Generated oracles**
twilio/twilio-java	25%	3%	15%	25%
bkiers/liqp	42%	7%	38%	42%
wmixvideo/nfe	61%	45%	79%	80%
wearefrank/ladybug	52%	19%	41%	52%
Average	45%	19%	43%	50%

RQ4 Findings: The generated oracles that compile and pass raise the mutation scores of the test suites with no oracles by 24.8% percentage points on average. Even with compilation failures and occasional false positives, LLM-generated oracles provide a substantial test suite improvement.

V. Discussion

In this paper we report the results of an empirical study about the effectiveness of LLMs to generate test oracles, by focusing on the *size of the LLM*, *the contextual information in the prompts*, and the effectiveness of the generated oracles in terms of *mutation analysis*. Here we summarize the main results and highlight concrete takeaways.

A. The Model Size Impacts the Oracle Accuracy (RQ1)

Findings: The accuracy of the generated oracles improves from around 3% to 29%, when we move from a 1.5-billion-parameter checkpoint to a 32-billion-parameter checkpoint.

Implications: The pronounced size effect suggests that oracle generation is capacity-limited rather than knowledge-limited. Favoring larger models over prompt engineering significantly widens the accuracy delta.

B. Code Models do not Significantly Impact the Oracle Accuracy (RQ1)

Findings: LLMs trained on code do not perform significantly better than LLMs trained on general text. The benefit appears logarithmic.

Implications: Favoring code LLMs over general LLMs does not significantly widen the accuracy delta.

C. A Test Prefix and Method Call Prompt Is Sufficient (RQ2)

Findings: Augmenting the prefix and method calls prompt with focal-class and test-class source increases context length, but does not significantly improve accuracy (less than one percent). The extra information adds latency and token cost without significant prediction quality gains.

Implications: Adding only the test prefix and the invoked methods minimizes latency and fits within context windows of many LLMs. The cost of extra information in the prompt does not pay off: the benefit is negligible while hitting model context length limits quickly.

D. LLM-Generated Oracles Boost the Mutation Score (RQ4)

Findings: The LLM-generated oracles largely boost the mutation score of the test suite (+25% on average over four projects) and achieve a mutation score comparable to carefully generated test oracles (-10% on average over four projects).

Implications: LLMs can generate surprisingly good test oracles for unit test cases, and can largely reduce human effort.

E. Open Research Directions

The results of the empirical study reported in this paper spotlight some key directions that can drive future research: Privilege the LLM size over code-specific training, focus on the quality of the test prefix and the code information rather than the amount of extra information provided with the prompt, and investigate how to generalize the good results of generating simple oracles towards the generation of particularly complex oracles.

VI. Threats to Validity

A. Internal Validity

The results may vary with different parameters and temperatures. We mitigated the risk of biased results by using only default LLM configurations, reporting the parameters and temperature used in the experiments, and making the benchmark publicly available in a replication package to allow the reproduction of the results.

The results may change with the future LLM milestones. We mitigated the risk of aging results by experimenting with 10 popular LLMs of different sizes, to investigate the impact of the LLM on the generation process. We experimented with open source LLMs only. Proprietary LLMs, like OpenAI's models, may perform better than the LLMs we evaluate in this paper.

The manual validation of the 13,866 generated oracles is almost impossible. We automatically validated the generated oracles by comparing them with the available and carefully designed oracles. We mitigated the risks of incorrect validation by implementing the comparison with string matching and inclusion that misses valid oracles that differ from the reference manual oracles only for even small syntactic elements. We decided to report conservative results that are a pessimistic under-approximation of the actual data, to avoid any risk of overestimating the results.

The test mining and mutation analysis setup may miss some test cases and mutants, thus reducing the effectiveness of the results. We rely on publicly available and well-known tools (PyDriller, Tree-Sitter, JavaParser, Maven, and PIT) to reduce the risk of mis-minings and mis-mutations due to errors in the tools.

B. External Validity

The results may be biased by the data used to build the benchmark and the building process.

We mitigated the risk of biased benchmarks by mining Java/Maven projects from popular repositories, and limiting the subjects to test cases added after September 2024, the last reported training cutoff date of the LLMs we used.

We experimented only with LLMs with a publicly declared cutoff date, and we executed all LLMs locally. We relied only on open-source LLMs for reproducibility. We excluded OpenAI models that are only available on the cloud and do not declare the cutoff date, to avoid the risk of biases due to executions out of our control.

VII. Related Work

In this paper, we propose the first large-scale evaluation of the capability of LLMs to generate test oracles, on an unbiased dataset intentionally designed to avoid leakage from the training set.

The publicly available evaluations of LLMs for generating test oracles appear in the still few papers that propose approaches that rely on LLMs to automatically generate test oracles. These papers evaluate the proposed approach on either public benchmarks or on small-scale datasets. Publicly available benchmarks are likely included in LLM training data and raise serious concerns about data leakage and inflated performance estimates.

The most popular benchmark, Defects4J [26], has been publicly available since 2014 and is likely included in LLM training data of approaches evaluated after 2020. Small benchmarks provide only preliminary evidence and cannot be generalized.

AthenaTest by Tufano et al. [24] is the first noteworthy approach to generate test cases with LLMs. AthenaTest extends the RNN-based approach of ATLAS [43] to generate the test prefix and oracle from the unit implementation and its context. The paper reports the results of the evaluation of AthenaTest on Defects4J [26].

TOGA by Dinella et al. [23] generates test oracles with a two-step neural ranking procedure. The paper evaluates the assertion oracle inference on a variant of the ATLAS [43] dataset and the exceptional oracle inference on a variant of Methods2Test [24]. The ATLAS dataset is a corpus collected from 9k open-source Java projects on GitHub. The variant of ATLAS used for supervised training of TOGA includes over 170,000 labeled samples. The paper evaluates the assertion oracle inference of TOGA on an ATLAS held-out test set of size 8,024 and the exceptional oracle inference on a held-out test set of Methods2Test of size 53,705. The dataset used for evaluating the assertion oracle inference is relatively small (size 8,024). The paper evaluates only TOGA, and the datasets are not publicly available for further comparison, according to the information of the authors of this paper.

Tratto by Molinelli et al. [16] generates axiomatic oracles with a neuro-symbolic approach that combines neural networks with symbolic analysis. The paper evaluates the approach on a subset of oracles from Defects4J [26] compatible with the dataset used to evaluate Jdoctor [7], to comparatively evaluate the LLM approach of Tratto with the NLP approach of Jdoctor. The comparison with Jdoctor indicates the potential of the approach, however, the use of a subset of oracles from Defects4J limits the generalizability of the results.

Hossain and Dwyer [17] analyze seven LLMs proposed before *ChatGPT* (before 2022) and TOGA with Defects4J [26]. Thus the evaluation suffers from the limitations of a dataset that has been publicly available for many years.

Khandaker et al. [38] compare test oracles generation on 9 different combinations of 3 general purpose LLMs (GPT4o, Hermes, and Llama) and 4 prompt types (including only the test prefix with, extending the contextual information to the methods of the focal class, and experimenting with the RAG approach [44] on both the simple and extended versions). They limit the analysis to general purpose LLMs, excluding those pre-trained on the code, the instruct and the reasoning models.

Wang et al.'s recent survey [30] confirms the absence of large-scale evaluation to assess the effectiveness of LLMs to generate assertions for test cases added after their training cutoff, and highlights the critical gap in understanding the true generalization ability of LLMs for generating oracles.

In this paper, we evaluate the state-of-the-art LLMs with a large and unbiased dataset intentionally designed to avoid leakage from the training set, and we make the dataset publicly available in a replication package for future evaluations, while all the datasets, except for Defects4J, are not publicly available, according to public information.

VIII. Conclusion

In this paper, we present the first unbiased study on the effectiveness of LLMs to generate test oracles. We show the experimental setting and the benchmark that we created to evaluate LLMs. The benchmark includes 13,866 test oracles that we mined from 135 Java projects, and that were created after the cut-off dates of the training of the LLMs used in the experiments, and are thus unbiased.

We discuss the results of the experiments that provide three surprising insights: (i) LLMs generate test oracles with an increase of the mutation score of the test cases comparable to carefully manually generated test oracles, (ii) the code LLM models do not perform significantly better than general models, (iii) the test prefix and the methods called in the program under test provide sufficient information to generate good oracles, while additional code context does not bring relevant benefits.

The results confirm the expectation that the size of the LLM matters and the difficulty of LLMs to generate complex test oracles, thus opening important research directions.

IX. Acknowledgment

This material is based upon work supported by SNF project A-Test Autonomic Software Testing (200021_215487), NSF grant CNS-2120070, and the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590132. Alberto Martin-Lopez is funded by an SNSF Ambizione grant (No. 233205).

References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE TSE*, vol. 41, no. 5, pp. 507–525, 2015.

[2] G. Fraser and A. Arcuri, “Whole test suite generation,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[3] A. Bertolino, P. Braione, G. D. Angelis, L. Gazzola, F. M. Kifetew, L. Mariani, M. Orrù, M. Pezzè, R. Pietrantuono, S. Russo, and P. Tonella, “A survey of field-based testing techniques,” *ACM Comput. Surv.*, vol. 54, no. 5, pp. 92:1–92:39, 2022. [Online]. Available: https://doi.org/10.1145/3447240

[4] M. Smytzek, M. Eberlein, B. Serçe, L. Grunske, and A. Zeller, “Tests4Py: A benchmark for system testing,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 557–561. [Online]. Available: https://doi.org/10.1145/3663529.3663798

[5] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” in *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Szeged, Hungary, Sep. 2011, pp. 416–419.

[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 75–84.

[7] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. Delgado Castellanos, “Translating code comments to procedure specifications,” in *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands, July 2018.

[8] R. Pandita, K. Taneja, L. Williams, and T. Tung, “ICON: Inferring temporal constraints from natural language API descriptions,” in *ICSM 2016: 32nd IEEE International Conference on Software Maintenance*, Raleigh, NC, USA, Oct. 2016, pp. 378–388.

[9] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, “C2S: Translating natural language comments to formal program specifications,” in *ESEC/FSE*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 25–37. [Online]. Available: https://doi.org/10.1145/3368089.3409716

[10] “The Llama 4 herd,” https://ai.meta.com/blog/llama-4-multimodal-intelligence/, accessed: May 30, 2025.

[11] DeepSeek-AI and et al., “DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning,” 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[12] M. Pezzè, S. Abrahão, B. Penzenstadler, D. Poshyvanyk, A. Roychoudhury, and T. Yue, “A 2030 roadmap for software engineering,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, June 2025. [Online]. Available: https://doi.org/10.1145/3731559

[13] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, “On the evaluation of large language models in unit test generation,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1607–1619. [Online]. Available: https://doi.org/10.1145/3691620.3695529

[14] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, “LLM-based test-driven interactive code generation: User study and empirical evaluation,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 9, p. 2254–2268, Sep. 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3428972

[15] L. Baresi and M. Young, “Test oracles,” U. of Oregon, Dept. of Computer Science, Tech. Rep. CIS-TR-01-02, 2001.

[16] D. Molinelli, A. Martin-Lopez, E. Zackrone, B. Eken, M. D. Ernst, and M. Pezzè, “Tratto: A neuro-symbolic approach to deriving axiomatic test oracles,” *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2025.

[17] S. B. Hossain and M. Dwyer, “TOGLL: Correct and strong test oracle generation with LLMs,” in *ICSE 2025, Proceedings of the 43rd International Conference on Software Engineering*, Ottawa, Ontario, Canada, Apr. 2025.

[18] A. Blasi, A. Gorla, M. D. Ernst, and M. Pezzè, “Call Me Maybe: Using NLP to automatically generate unit test cases respecting temporal constraints,” in *ASE 2022: Proceedings of the 37th Annual International Conference on Automated Software Engineering*, Oakland Center, MI, USA, Oct. 2022, pp. 1–11.

[19] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, “MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation,” *J. Sys. Softw.*, vol. 181, pp. 111 041:1–13, Nov. 2021.

[20] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, “@tComment: Testing Javadoc comments to detect comment-code inconsistencies,” in *ICST 2012: Fifth International Conference on Software Testing, Verification and Validation (ICST)*, Montreal, Canada, Apr. 2012, pp. 260–269.

[21] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[22] D. K. Peters and D. L. Parnas, “Using test oracles generated from program documentation,” *IEEE Transactions on software engineering*, vol. 24, no. 3, pp. 161–173, 2002.

[23] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, “TOGA: a neural method for test oracle generation,” in *ICSE 2022, Proceedings of the 43rd International Conference on Software Engineering*, Pittsburgh, PA, USA, May 2022, pp. 2130–2141.

[24] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, “Unit test case generation with transformers and focal context,” https://arxiv.org/abs/2009.05617, May 2021.

[25] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, “Generating accurate assert statements for unit test cases using pretrained transformers,” in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 54–64. [Online]. Available: https://doi.org/10.1145/3524481.3527220

[26] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA, July 2014, pp. 437–440, tool demo.

[27] A. Abdullin, P. Derakhshanfar, and A. Panichella, “Test wars: A comparative study of SBST, symbolic execution, and LLM-based approaches to unit test generation,” *arXiv preprint arXiv:2501.10200*, 2025.

[28] K. Zhou, Y. Zhu, Z. Chen, W. Chen, W. X. Zhao, X. Chen, Y. Lin, J. Wen, and J. Han, “Don’t make your LLM an evaluation benchmark cheater,” *CoRR*, vol. abs/2311.01964, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2311.01964

[29] R. Xu, Z. Wang, R. Fan, and P. Liu, “Benchmarking benchmark leakage in large language models,” *CoRR*, vol. abs/2404.18824, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.18824

[30] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software testing with large language models: Survey, landscape, and vision,” *IEEE Transactions on Software Engineering*, 2024.

[31] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, “Perfect is the enemy of test oracle,” in *Proceedings of the 30th acm joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 70–81.

[32] “JavaParser,” https://javaparser.org/, [Accessed 25-08-2025].

[33] “Introduction - Tree-sitter — tree-sitter.github.io,” https://tree-sitter.github.io/tree-sitter/, [Accessed 15-09-2025].

[34] “Pydriller,” https://pypi.org/project/PyDriller/, [Accessed 25-08-2025].

[35] “Ollama — ollama.com,” https://ollama.com/, [Accessed 25-08-2025].

[36] “vLLM — docs.vllm.ai,” https://docs.vllm.ai/en/latest/, [Accessed 25-08-2025].

[37] O. Dabic, E. Aghajani, and G. Bavota, “Sampling projects in GitHub for MSR studies,” in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.

[38] S. M. Khandaker, F. M. Kifetew, D. Prandi, and A. Susi, “AugmenTest: Enhancing tests with LLM-driven oracles,” in *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 2025, pp. 279–289. [Online]. Available: https://doi.org/10.1109/ICST62969.2025.10988926

[39] Y. Kuratov, A. Bulatov, P. Anokhin, I. Rodkin, D. Sorokin, A. Sorokin, and M. Burtsev, “BABILong: Testing the limits of LLMs with long context reasoning-in-a-haystack,” 2024. [Online]. Available: https://arxiv.org/abs/2406.10149

[40] L. Di Grazia and M. Pradel, “Code search: A survey of techniques for finding code,” *ACM Comput. Surv.*, vol. 55, no. 11, Feb. 2023. [Online]. Available: https://doi.org/10.1145/3565971

[41] C. Clement, S. Lu, X. Liu, M. Tufano, D. Drain, N. Duan, N. Sundaresan, and A. Svyatkovskiy, “Long-range modeling of source code files with eWASH: Extended window access by syntax hierarchy,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 4713–4722.

[42] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Tang, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, “Qwen2.5 technical report,” https://arxiv.org/abs/2412.15115, Jan. 2025.

[43] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, “On learning meaningful assert statements for unit test cases,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[44] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, Q. Guo, M. Wang, and H. Wang, “Retrieval-augmented generation for large language models: A survey,” *CoRR*, vol. abs/2312.10997, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.10997