Understanding LLM-Driven Test Oracle Generation

Adam Bodicoat*, Gunel Jahangirova†, and Valerio Terragni*
*University of Auckland, Auckland, New Zealand
Email: abod278@aucklanduni.ac.nz, v.terragni@auckland.ac.nz
†King's College London, London, United Kingdom
Email: gunel.jahangirova@kcl.ac.uk

***Abstract*—Automated unit test generation aims to improve software quality while reducing the time and effort required for creating tests manually. However, existing techniques primarily generate regression oracles that predicate on the implemented behavior of the class under test. They do not address the oracle problem: the challenge of distinguishing correct from incorrect program behavior.**

With the rise of Foundation Models (FMs), particularly Large Language Models (LLMs), there is a new opportunity to generate test oracles that reflect intended behavior. This positions LLMs as enablers of Promptware, where software creation and testing are driven by natural-language prompts.

This paper presents an empirical study on the effectiveness of LLMs in generating test oracles that expose software failures. We investigate how different prompting strategies and levels of contextual input impact the quality of LLM-generated oracles. Our findings offer insights into the strengths and limitations of LLM-based oracle generation in the FM era, improving our understanding of their capabilities and fostering future research in this area.

***Index Terms*—Large Language Models, Foundation Models, Software Testing, Test Oracle Problem, Automated Test Generation, Prompt Engineering, Promptware, AI4SE**

I. Introduction

Software testing is crucial for ensuring software quality and reliability. However, manually creating test oracles is labor-intensive. While automated test generation tools like EvoSuite [1] and Randoop [2] typically rely on regression oracles, which assume the current version is correct [3], [4]. This limits their ability to detect faults in the given version, this is the long-standing **oracle problem** in test automation [5], [6], [3].

In the era of **Foundation Models (FMs)**, Large Language Models (LLMs) offer new opportunities to address this problem [7]. With their capabilities in natural language understanding, pattern recognition, and contextual reasoning, LLMs can help bridge the gap between automated test generation and meaningful oracle creation [8], [9], [10], [11], [12]. By interpreting developer intent through prompts, LLMs can generate oracles that align with expected behaviors rather than implemented ones [4], [13], [3], [14].

Recent research has started to explore LLMs for generating tests and oracles [13], [4], [15], [16], [9], [17], but critical questions remain. In particular, regarding how prompting strategies and input context affect oracle quality. Understanding these factors is essential as we transition toward **Promptware**, where software validation may be driven by natural-language prompts crafted by both developers and non-developer prompt experts.

To address this gap, this paper present an empirical study on how prompting techniques (zero-shot, few-shot, chain-of-thought [CoT], and tree-of-thoughts [ToT]) and contextual inputs (test prefix alone, test prefix plus method under test [MUT], and test prefix plus class under test [CUT]) influence the quality of LLM-generated test oracles. Our study isolates the oracle-generation capability of LLMs rather than the broader task of fault reproduction. We therefore do not include failure descriptions or bug reports in the prompt. Incorporating such inputs would shift the task toward bug reproduction or localization, which are orthogonal to the oracle problem. To the best of our knowledge, this is the first study to conduct such an analysis.

We used the **GitHub Recent Bugs (GHRB)** benchmark [18], a dataset designed to evaluate LLMs on real-world Java bugs while mitigating data leakage risks. For each bug, we provide to LLMs the buggy code and its triggering test input – omitting the original oracle – and analyze oracle quality based on compilation success and bug exposure. Our experiments involved 36 representative GHRB bugs and two LLMs, GPT-4o and StarCoder.

Our results reveal several important trends:

① Oracles generated with more context compile and detect bugs more reliably. CUT-level context significantly outperforms other configurations, achieving 53.64% accuracy versus 40.74% (MUT) and 40.38% (test prefix only). This is an expected result.

② Prompting style matters: zero-shot and few-shot prompts yield higher compilation rates (67.38% and 72.96%) and accuracy (54.56% and 51.30%) than CoT and ToT, which struggle with low compilation (both below 50%).

③ Incorporating the CUT in the input prompt, along with zero-shot and few-shot prompting techniques, leads to the most consistently accurate LLM-generated test oracles. However, our findings show there is potential for reasoning based prompt techniques like CoT and ToT to be able to produce accurate test oracles given their high accuracy when they do produce compilable assertions.

④ While StarCoder slightly outperforms GPT-4o in average accuracy, GPT-4o paired with CUT context delivers the most consistently accurate oracles across all combinations.

⑤ Prompting strategy has a stronger impact on oracle effectiveness than LLM choice.

These findings suggest that prompt design and context play a critical role in the effectiveness of LLM-based oracle generation. While reasoning-driven prompting (e.g., CoT, ToT) shows potential when it compiles, zero-shot and few-shot prompting currently offer the best tradeoff between accuracy and robustness. Our study offers guidance for AI-assisted testing tools usable by both testing and prompt experts in the FM era.

In summary, this paper makes the following contributions:

- An empirical study of how prompt types and contextual inputs affect LLM-generated test oracle quality.
- In support of Promptware, a series of insights into the effectiveness of LLMs in generating test oracles, providing guidance for the future of LLM-driven oracle generation.
- Public release of code to ensure reproducibility and support future research in this area [19].

II. Experimental Design

Our study aims to assess the ability of LLMs to generate accurate and correct test oracles using different input prompt variations. These variations fall into two categories: the content of the prompt and the prompting technique. Specifically, our empirical study investigates the following research questions:

RQ1 - Input Context What influence does the content and context within the prompt have on the accuracy of LLM generated test oracles?

RQ2 - Prompt Engineering How do different prompt engineering techniques impact the accuracy of LLM generated test oracles relative to each other?

RQ3 - Model Comparison How do different LLMs impact the accuracy of LLM generated test oracles?

RQ4 - Impact Analysis Which factor has the most significant impact on LLM-generated test oracles?

RQ1 studies how the content of the prompt influences LLM-generated test oracles. This helps determine what content provides the best context for test oracle generation. We use three levels of context: Test Prefix, Test Prefix with Method Under Test, and Test Prefix with Class Under Test.

RQ2 investigates the impact the prompting technique has on the accuracy of LLM generated test oracles. This allows us to compare the effectiveness of different prompting techniques in generating test oracles from existing test prefixes. In this study, we will be using four prompting techniques: zero shot, few shot, chain of thought, and tree of thoughts.

RQ3 examines how different LLMs impact test oracle generation based on their training differences.

RQ4 analyses the impacts the prompt content, prompting technique, and LLM have on the accuracy of the generated test oracles relative to each other. This allows us to determine the relative importance of each of these factors for LLM driven test oracle generation. In this study, we will measure the impact each variable we study has using the range of average accuracies for that variable. We also compare each result to find the factors which contribute to the highest average accuracy.

To answer these RQs, we conducted an experiment asking the selected LLMs to generate test oracles for each prompt, where the prompt is a unique combination of the prompt context and prompting technique. The test cases used fail on the buggy version of the code but pass on the corrected version. We remove the test oracle and provide the modified test case and the buggy code as input to the LLM. We then prompt the LLM to generate a suitable test oracle. We evaluate the correctness based on whether the generated oracle fails the buggy version and passes the correct version. This allows us to analyse how prompt content, prompting technique, and LLM influence oracle performance.

A. Dataset

This study uses the **GitHub Recent Bugs (GHRB)** [18] benchmark, a dataset designed for evaluating the performance of LLMs on real world code. GHRB ensures bugs and fixes come from real repositories updated after the training period of popular LLMs, including StarCoder and GPT-4o. This avoids data leakage and model memorisation [18]. As of December 2024, GHRB contains 107 bugs from 16 popular open source Java repositories. We selected 36 bugs from this dataset, based on criteria designed to ensure experimental consistency and validity.

1) Absence of compile errors: Both the buggy and correct versions of the selected repositories were verified to be free of compile-time errors. This criterion was critical for eliminating confounding factors that could distort the evaluation of the generated test oracles. Ensuring compilability allowed the analysis to focus solely on the semantic correctness of the generated oracles. Compilation errors were likely due to incomplete fixes, missing dependencies, or build configuration issues present in certain repositories within the benchmark dataset.

2) Random sampling: To minimise selection bias and enhance the generalisability of the findings, the subset of bugs are randomly sampled from those without compilation errors in the benchmark [20]. This approach ensures that the selected bugs represent a diverse and unbiased subset of the dataset [21]. We choose to select 36 bugs only due to the high computation cost required for larger subsets.

Similar to established bug benchmarks such as Defects4J [22], each repository in the dataset includes a buggy version, where bug-revealing test cases fail, and a corresponding correct version, where the same test cases pass. We also ensure that the bug requires a test assertion to be exposed and does not result in unwanted exceptions, which makes the test fails without the need of assertions.

Figures 1 and 2 illustrate a concrete example from the JSoup repository in the GHRB benchmark. Specifically, Figure 1 presents a bug-revealing test case designed to verify

```
1  @Test
2  public void testCopyConstructor_noSideEffectOnAttributes()↩
       {
3      Safelist safelist1 = Safelist.none().addAttributes(↩
           TEST_TAG, TEST_ATTRIBUTE);
4      Safelist safelist2 = new Safelist(safelist1);
5      safelist1.addAttributes(TEST_TAG, "invalidAttribute");
6
7      assertFalse(safelist2.isSafeAttribute(TEST_TAG null new
           Attribute("invalidAttribute" TEST_VALUE)));
8  }
```

Fig. 1. Bug-revealing test case exposing incorrect copy behavior in Jsoup's `Safelist` class (GHRB benchmark)

the correctness of the copy constructor in the `Safelist` class. The test case first creates an instance of the `Safelist` (`safelist1`), copies it into a second instance (`safelist2`), and subsequently modifies the original object by adding an additional attribute (`"invalidAttribute"`). The assertion (highlighted in violet) explicitly checks that this newly added attribute in the original object should *not* be recognized as safe in the copied instance, confirming that the copy constructor correctly creates an independent copy.

However, the buggy implementation of the copy constructor (shown in Figure 2) fails to perform a deep copy. Instead, it simply reuses references to internal collections (`attributes`, `tagNames`, etc.), leading to unintended side effects. As a result, any subsequent changes made to the original instance incorrectly propagate to the copied instance, causing the test case to fail. The goal of our study is to investigate the key factors influencing the effectiveness of LLMs in generating test oracles that accurately detect software faults.

B. Prompt Construction

This study evaluates the impact of four prompting techniques on test oracle generation: zero shot, few shot, chain of thought, and tree of thoughts. These were chosen for their ability to capture different dimensions of model reasoning and generalization [23].

Zero-Shot prompting (Z) involves asking the model to perform a task without providing any prior examples or structured guidance [24]. For test oracle generation, this technique entails supplying the model with a task description to generate test oracles for the given test prefix. This approach evaluates the model's inherent understanding of the task based solely on the prompt content and the model's pre-training.

Few-Shot prompting (F) involves providing the model with a limited number of examples illustrating the task at hand [24], [25]. In this study, we use three test oracle examples from the same repository as the test prefix to balance prompt length with relevant context [24], [25]. Using examples from the same repository ensures consistency, and the same examples are used for all bugs in that repository to reduce variability and focus on prompt variations.

Chain of Thought (CoT) prompting (Ch) guides the model through a structured reasoning process [26]. Following prior work [25], the CoT prompt for generating oracles includes the following steps:

```
1  /**
2   Deep copy an existing Safelist to a new Safelist.
3   @param copy the Safelist to copy
4   */
5  public Safelist(Safelist copy) {
6      this();
7      tagNames.addAll(copy.tagNames);
8      attributes.putAll(copy.attributes);
9      enforcedAttributes.putAll(copy.enforcedAttributes);
10     protocols.putAll(copy.protocols);
11     preserveRelativeLinks = copy.preserveRelativeLinks;
12 }
```

Fig. 2. Buggy implementation of the `Safelist` copy constructor causing unintended side-effects (GHRB benchmark)

- Identifying the purpose of the test prefix and code under test (if provided)
- Determining the expected outcome of the code and test
- Formulating the correct test oracles

This approach is designed to encourage logical reasoning to improve the accuracy of the generated oracles.

Tree of Thoughts (ToT) prompting (Tr) extends chain of thought reasoning by exploring multiple potential paths to solve the problem before converging on the most plausible solution [27]. As per previous work [28], in the context of test oracle generation, the prompt includes the following steps to construct a ToT prompt:

- Root thought: Initial understanding of the test prefix and code under test (if provided).
- Branching: Exploring different paths of reasoning.
- Expansion: Developing each reasoning path further.
- Pruning: Removing invalid or redundant paths.
- Combinations: Combining insights from valid branches into a set of coherent and accurate test oracles.

This method leverages the model's ability to consider diverse reasoning paths, potentially improving robustness in scenarios with complex or ambiguous prompts.

To further analyse the impact of different prompt configs, each prompting technique was tested with three levels of **input content**:

Test Prefix (T): The minimal context consisting of the test code without the test oracles. Figure 1 is an example of a bug revealing test that is used in the prompt input but with the test oracle on line seven removed.

Test Prefix + MUT (M) : Adds the method being tested to provide additional context for the oracle generation. Figure 2 is an example of the method under test to be added to the prompt input corresponding to the test case in Figure 1.

Test Prefix + CUT (C): Adds the class containing the test and method, providing more comprehensive context.

In the prompts, all test assertion oracles are removed from the test prefix, which may contain multiple assertions. The prompt let the LLM decide how many assertions should be generated. The provided CUT and MUT come from the buggy version, but we do not indicate in the prompt that the test exposes a bug. This reflects a realistic scenario where the goal is to generate an oracle without prior knowledge of whether the test reveals a bug. We refine the prompt iteratively, following

established prompt engineering guidelines and prior work on LLM-driven test oracle generation [13], [15], [4], [16], [9], [29]. We manually identified MUT and CUT by examining the code changes in the corresponding corrected version in the GHRB repository. *Due to space constraints, prompts are omitted but available in the supplementary material [19]*

C. Models

For this study, we selected two LLMs: STARCODER and GPT-4O, enabling comparison between a domain-specific and a general-purpose model regarding prompt effects on test oracle generation. These models were chosen for their popularity and because the GHRB benchmark avoids data leakage in them [18].

STARCODER (S) (v. 1.0) is a domain-specific LLM for code completion, trained on a large code corpus as part of the BigCode Project [30]. Its focus on code generation makes it well suited for generating test oracles from test prefixes.

GPT-4O (G) (`gpt-4o-mini-2024-07-18`) is a general-purpose LLM trained on diverse data across domains (including source code) [31], enabling it to handle tasks like code generation.

The inclusion of GPT-4O allows for an exploration of how prompt input influences oracle generation in a general-purpose model, providing a useful benchmark for comparing the performance of specialised and non-specialised models.

While other advanced reasoning-oriented models, such as GPT-4O1, are available, we intentionally selected STARCODER and GPT-4O due to their documented effectiveness in code-related tasks [30]. Although including reasoning-focused LLMs might offer additional insights, it remains uncertain whether using these newer models is feasible. The GHRB benchmark was specifically designed to prevent data leakage in STARCODER and GPT-4O, but it does not explicitly account for more recent models [18]. Moreover, STARCODER is a state-of-the-art model specialized for coding tasks, while GPT-4O is commonly used as a baseline in code generation.

D. Experimental Setup

We implement an automated framework to run our experiments. It first runs tests on the buggy and correct GHRB repositories and records which tests fail or pass to confirm expected behavior. It then removes the test oracle from the bug-revealing test in both versions and appends the oracleless test, along with other relevant information, to the LLM input. The framework inserts the LLM-generated oracles into both versions, compiles, and runs the tests. It checks whether tests compile and which tests pass or fail, comparing results to the original output to assess accuracy and correctness. Each experiment is repeated five times to account for variability in LLM outputs.

This controlled configuration simulates a workflow in which an automated test generator (e.g., EVOSUITE [32]) produces numerous assertion-free tests. The assertions typically generated by EVOSUITE are regression oracles that replicate existing behavior and are therefore not suitable for detecting faults in the current version. In our setup, the LLM complements such tools by generating new, potentially fault-revealing assertions. Similar hybrid pipelines have been explored in prior work, such as TOGLL [13]. Hence, although our setup is artificial, it closely mirrors a plausible integration of LLMs within modern automated testing workflows.

Each experiment combines an LLM, prompting technique, and prompt content. Since STARCODER is code-only [30], it supports zero-shot and few-shot prompting. GPT-4O, with NLP capabilities, also supports CoT and ToT [33]. This results in $\mathbf{2{,}160}$ **total runs** = 36 bugs $\times$ 3 prompt contexts $\times$ $(2+4)$ techniques $\times$ 5 repetitions.

In all experiments, we set the LLM `temperature` to 0 and `top-p` to 1, following prior studies [34], [35], [36]. Indeed, lower temperatures are often preferable when generating code [37].

E. Metrics

We evaluated the accuracy of the generated test oracles by comparing the outputs of tests with the replaced oracles to those with the original oracles on both the buggy and correct versions [38]. To assess correctness, we measured the following:

- Accuracy: Tests correctly differentiating between buggy and correct versions: The number of cases in which the oracle(s) demonstrated consistent behaviour by correctly failing for the buggy version and passing for the correct version of the code. This is the optimum result.

- Buggy Accuracy: Tests correctly failing on buggy version: The number of test cases where the generated oracle(s) correctly identified a failure in the buggy version of the code.

- Correct Accuracy: Tests correctly passing on correct version: The number of test cases where the generated oracle(s) correctly passed the relevant tests in the correct version of the code.

- Compilation Rate: The rate at which generated test oracles compile.

These metrics evaluate the accuracy and performance of LLM-generated oracles, enabling a comparative analysis across various prompt configurations.

III. RESULTS

A. RQ1 - Input Context

RQ1 : *What influence does the content and context within the prompt have on the accuracy of LLM generated test oracles?*

Table I presents the average accuracy across all experiments for all 36 bugs for each prompt content scenario studied. We report the average across five runs, as the observed variation between runs was minimal. The results indicate that providing more context in the prompt improves compilation

TABLE I
PROMPT CONTENT COMPARISON (RQ1)

Content	Acc.%	Buggy Acc.%	Correct Acc.%	Comp. Rate%
Test Prefix	40.74	47.78	42.22	49.36
Prefix + MUT	40.38	53.08	43.85	59.23
Prefix + CUT	53.64	70.84	58.13	76.26

rates. Specifically, prompts containing only the test prefix achieve an average compilation rate of 49.36%, while adding the MUT increases this rate to 59.23%, and incorporating the CUT raises it further to 76.26%.

Table I also shows that the LLM-generated oracles cause buggy versions to fail correctly more often than they enable correct versions to pass correctly. For instance, with just the test prefix, buggy versions fail correctly 47.78% of the time, compared to passing 42.22% of the time for correct versions. When the MUT is included, these rates increase to 53.08% and 43.85%, respectively. Incorporating the CUT results in further improvements, with buggy versions failing correctly at 70.84% and correct versions passing correctly at 58.13%. These findings suggest that while LLMs can generate bug-revealing oracles, the generated oracles may still incorrectly fail on the correct versions of the code, indicating that some identified bugs may not align with the expected program behaviour.

The results show that adding the CUT yields a greater improvement than adding the MUT. Transitioning from the test prefix to the MUT increases the buggy version failure rate by 5.30% and the correct version pass rate by 1.63%. However, moving from the MUT to the CUT leads to much larger gains: a 17.76% increase in the buggy failure rate and a 14.28% increase in the correct pass rate. These findings indicate that while the MUT offers some useful context, the CUT provides far more critical information, greatly enhancing the LLM's ability to generate accurate test oracles.

When considering overall accuracy, test oracles generated with only the test prefix achieve 40.74%, which slightly decreases to 40.38% when the MUT is added. However, including the CUT leads to a significant increase, with accuracy reaching 53.64%. The negligible change between the test prefix and MUT scenarios, compared to the substantial improvement when introducing the CUT, reinforces the conclusion that greater contextual information in prompts enhances the LLM's ability to generate correct and accurate test oracles. This result suggests that the absence of CUT context may hinder the LLM's understanding of the code under test, reducing the quality of the generated oracles.

> **Answering RQ1:** LLMs generate compilable and accurate test oracles more consistently when the prompt includes the CUT.

B. RQ2 - Prompt Engineering

RQ2 : *How do different prompt engineering techniques impact the accuracy of LLM generated test oracles relative to each other?*

TABLE II
PROMPTING TECHNIQUE COMPARISON (RQ2)

Technique	Acc.%	Buggy Acc.%	Correct Acc.%	Comp. Rate%
Zero-Shot	54.56	64.47	57.28	67.38
Few-Shot	51.30	68.33	55.00	72.96
CoT	31.11	38.52	32.59	44.44
ToT	29.26	41.11	32.22	44.81

Table II presents the average accuracy across all bugs for each prompting technique studied. We report the average across five runs, as the observed variation between runs was minimal. The results demonstrate that few-shot and zero-shot prompting techniques produce LLM-generated oracles with higher compilation rates compared to CoT and ToT techniques. Specifically, zero-shot scenarios achieve a compilation rate of 67.38%, few-shot scenarios reach 72.96%, while CoT and ToT techniques result in lower compilation rates of 44.44% and 44.81%, respectively.

The data also reveal a significant increase in the rate of buggy versions failing correctly compared to correct versions passing correctly across all prompting techniques. For zero-shot prompting, the rate increases from 57.28% for correct versions passing to 64.47% for buggy versions failing correctly. Similarly, for few-shot prompting, the rate rises from 55.00% to 68.33%. CoT and ToT prompting exhibit smaller but still notable increases with CoT going from 32.59% to 38.52% and ToT from 32.22% to 41.11%. These results, consistent with the findings from *RQ1*, suggest that LLM-generated test oracles may not fully align with the expected behaviour, leading to failures on both buggy and correct versions of the code.

Zero-shot prompting achieves the highest accuracy at 54.56%, followed by few-shot prompting at 51.30%. CoT and ToT show significantly lower accuracies at 31.11% and 29.26%, respectively. The slight drop from zero-shot to few-shot may result from examples in few-shot prompts leading the LLM to deviate from expected behavior. The low performance of CoT and ToT may stem from their broader reasoning scope, which can cause the LLM to generate oracles that address general scenarios rather than the specific test prefix.

However, when just considering compiling oracles, zero-shot scenarios have an accuracy of 80.97%, few-shot scenarios with 70.31%, CoT scenarios with 70.00%, and ToT scenarios with 65.30%. This shows, when generating oracles that compile, while still demonstrating worse accuracy than zero-shot and few-shot scenarios, CoT and ToT scenarios are not significantly less accurate as the raw accuracies would suggest. Therefore, alongside the broadened scope for CoT and ToT prompts, a major issue is ensuring the LLM produces compilable oracles using these prompting techniques.

> **Answering RQ2:** LLMs more consistently generate compilable and accurate test oracles with zero-shot and few-shot prompting technique compared to reasoning based techniques like CoT and ToT.

TABLE III
LLM COMPARISON – ZERO-SHOT AND FEW-SHOT (RQ3)

LLM	Acc%	Buggy Acc.%	Correct Acc.%	Comp. Rate%
STARCODER	56.31	79.42	61.36	83.69
GPT-4O	49.63	54.07	51.11	57.41

TABLE IV
LLM COMPARISON – ALL PROMPTS (RQ3)

LLM	Acc.%	Buggy Acc.%	Correct Acc.%	Comp. Rate%
STARCODER	56.31	79.42	61.36	83.69
GPT-4O	39.54	46.94	41.76	51.02

C. RQ3 - Model Comparison

RQ3 : *How do different LLMs impact the accuracy of LLM generated test oracles?*

Table III shows the average results for each LLM using zero-shot and few-shot prompts. STARCODER achieves a much higher average compilation rate than GPT-4O, which is expected given STARCODER's focus on code generation [30], unlike the more general-purpose GPT-4O. This limits GPT-4O's performance, with an average accuracy of 49.63% compared to STARCODER's 56.31%. STARCODER also outperforms GPT-4O in both average buggy and correct accuracy for these prompting techniques.

Table IV includes the CoT and ToT scenarios used with GPT-4O which could not be used with STARCODER. Here, we see the CoT and ToT prompts significantly decrease GPT-4O's average compilation rate and accuracies – GPT-4O's average compilation rate and accuracy are reduced to 51.02% and 39.54% respectively.

However, when considering only test oracles that successfully compile, GPT-4O demonstrates a competitive capacity for generating accurate oracles. Specifically, the accuracy of GPT-4O's compiling test oracles is 86.45% compared to STARCODER's 67.28% when considering zero-shot and few-shot scenarios. Even when including compiling oracles from CoT and ToT prompts, GPT-4O achieves an accuracy of 77.50%. Though lower than zero-shot and few-shot accuracy, this shows GPT-4O's potential for reliable oracle generation in specific conditions.

> **Answering RQ3:** STARCODER-generated test oracles demonstrate a higher average compilation rate than GPT-4O generated ones resulting in higher accuracies.

D. RQ4 -Impact Analysis

RQ4 : *Which factor has the most significant impact on LLM generated test oracles?*

Table I shows that including the test prefix and the CUT has the highest average accuracy of 53.64% and including the test prefix and the MUT has the lowest average accuracy of 40.38%, giving a range of 13.26%. Considering the prompting technique, from Table II, zero-shot has the the highest accuracy of 54.56% and ToT has the lowest accuracy of 29.26%, giving a range of 25.30%. Considering the LLM used, from Table IV, STARCODER has the highest accuracy of 56.31% and GPT-4O has the lowest accuracy of 49.63% for cases where the same prompting techniques are used, giving a range of 6.68%. These findings indicate the prompting technique is the most significant factor out of these because of the far greater range of results observed compared to the content and LLM. This is followed by the prompt content, and then the LLM.

TABLE V
AVERAGE RESULTS FOR EACH CONFIGURATION (RQ4)

Config	Acc. Rank	Acc.%	Buggy Acc.%	Correct Acc.%	Comp. Rate%
S.Z.T.	4	66.67	77.78	66.67	77.78
S.Z.M.	3	68.57	84.29	71.43	88.57
S.Z.C.	6	55.29	88.24	60.00	92.94
S.F.T.	5	55.56	66.67	55.56	66.67
S.F.M.	9	44.44	77.78	55.56	83.33
S.F.C.	7	50.00	83.33	61.11	94.44
G.Z.T.	18	22.22	22.22	31.11	31.11
G.Z.M.	10	44.44	44.44	44.44	44.44
G.Z.C.	2	73.33	75.56	73.33	75.56
G.F.T.	12	37.78	44.44	37.78	44.44
G.F.M.	8	45.56	51.11	45.56	62.22
G.F.C.	1	74.44	86.67	74.44	86.67
G.Ch.T.	15	26.67	33.33	26.67	33.33
G.Ch.M.	17	22.22	28.89	28.89	44.44
G.Ch.C.	11	40.00	53.33	42.22	55.56
G.Tr.T.	13	35.56	42.22	35.56	42.22
G.Tr.M.	16	23.33	38.89	23.33	38.89
G.Tr.C.	14	28.89	42.22	37.78	53.33

Table V ranks each scenario's average accuracy. Column "Config" represents the `LLM. Prompt-Strategy. InputContent` setup (e.g., *G.Ch.C.* denotes *GPT, CoT, and Text Prefix + CUT*). We have introduced the Labels when describing the components individually (e.g., Text Prefix + CUT (**C**) in Section II).

Column "Acc. Rank" gives the ranking of the configurations ordered by Column 'Acc.%'. From this column we can calculate average rank of each different component. Out of the 18 scenarios conducted, the average rank of STARCODER scenarios is 5.67 and the average rank of GPT-4O scenarios is 11.42. The average rank of zero-shot is 7.17, few-shot is 7.00, CoT is 14.33, and ToT scenarios is 14.33 too. The average rank for just test prefix scenarios is 11.17, for test prefix and MUT scenarios is 10.5, and for test prefix and CUT scenarios is 6.83. Because STARCODER is not tested with CoT and ToT techniques, if we remove these six scenarios, the average rank of STARCODER is 5.67, GPT-4O is 7.33, zero-shot scenarios is 6.17, few-shot scenarios is 6.83, test prefix scenarios is 8, test prefix and MUT scenarios is 7.5, and test prefix and CUT scenarios is 4.00.

Figure 3 ranks the data from Table V in descending order by average accuracy. From this, we compute the average rank of each factor across the 18 combinations. STARCODER scenarios rank highest with an average of 5.67, followed by prompts with the CUT at 6.83, and few-shot prompts at 7. In contrast, CoT and ToT have the lowest average rank at 14.33. These results suggest that prompting technique is the most influential factor, as it shows the greatest variation in performance across the rankings.

We expect STARCODER to rank among the highest in

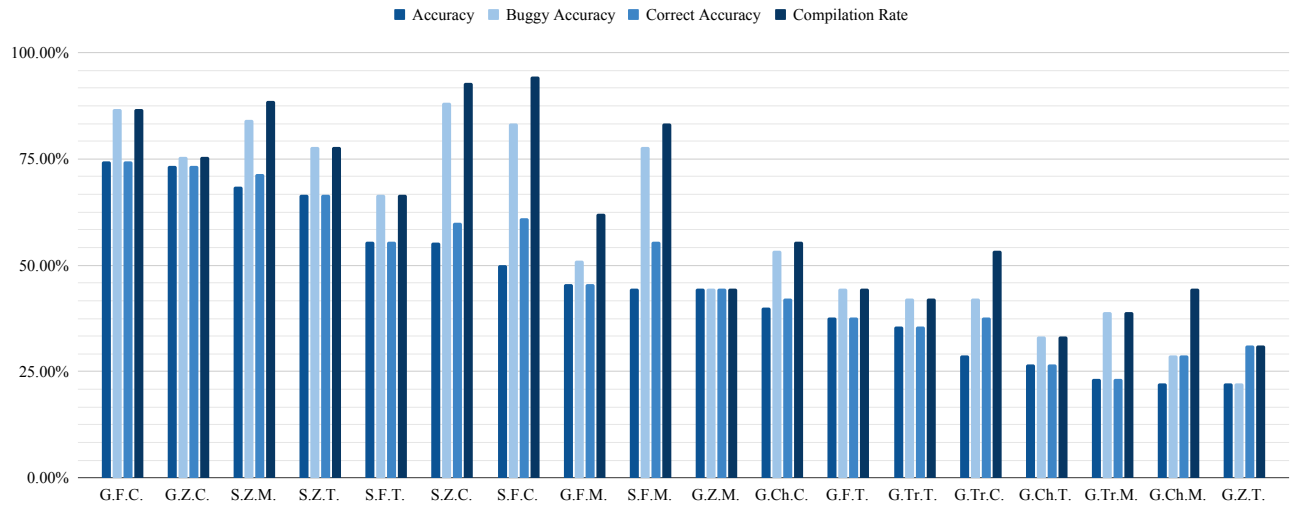

Fig. 3. Bar chart of results from all experiments ordered by average accuracy grouped by configuration (RQ4)

accuracy due to its code-focused training and exclusion from CoT/ToT prompts. Without these scenarios, STARCODER still outperforms GPT-4O (5.67 vs. 7.33), but CUT scenarios now have the highest rank (4), highlighting their importance for generating accurate test oracles.

> **Answering RQ4:** The prompting technique has the greatest impact on the accuracy of LLM-generated test oracles.

IV. DISCUSSION

A. Improvements with the CUT (RQ1)

The findings for *RQ1* show that including the CUT significantly improves both accuracy and compilation rates of LLM-generated oracles. While adding the MUT boosts compilation over the test prefix alone, likely due to added method context, it results in lower overall accuracy. This suggests the MUT provides limited guidance, whereas the CUT offers essential functional context and usage patterns for accurate oracle generation.

B. Prompting Strategies (RQ2)

For *RQ2*, it is evident that zero-shot and few-shot prompting scenarios outperform CoT and ToT strategies across all metrics. We attribute this to CoT and ToT encouraging broader reasoning, which often seem to lead to oracles that include irrelevant or unintended behavior. This findings suggest that while useful for general reasoning [25], these strategies may hinder concise, behavior-specific oracle generation. Future work could combine CoT/ToT with few-shot examples aligned with expected behaviour to balance reasoning strength with improved accuracy and compilation.

C. Model-Specific Performance (RQ3)

The comparative analysis of STARCODER and GPT-4O in zero-shot and few-shot scenarios reveals key differences. STARCODER, trained exclusively on code, achieves higher accuracy and compilation rates than GPT-4O, producing compilable code more consistently [39]. However, GPT-4O better reflects expected program behaviour despite a lower compilation rate. Addressing these contrasting strengths and weaknesses could involve combining insights from both models to design prompting techniques or hybrid approaches that leverage STARCODER's compilation strength and GPT-4O's capacity for accurate behaviour representation.

D. Other Considerations (RQ4)

The findings from *RQ4* show that using STARCODER gives the most accurate test oracles overall. However, STARCODER was not tested with CoT and ToT, which lowers the average accuracy of the other scenarios since CoT and ToT perform poorly. When we remove CoT and ToT, we see that adding the CUT has a bigger positive effect on accuracy than using STARCODER.

From Figure 3, we see that the highest average accuracies come from using GPT-4O with the CUT. *RQ3* also shows that for compiling oracles, GPT-4O has higher average accuracy than STARCODER. This is clear in Figure 3, especially in the S.Z.C., S.F.C., and S.F.M. cases, where the oracles compile well but have lower accuracy. With more context, GPT-4O generates more accurate oracles than STARCODER, but with less context, STARCODER performs better. Thus, the best LLM depends on the prompt's level of context.

Although the evaluation scenario is controlled, it remains useful for understanding how prompting and context influence oracle synthesis. We assume the faulty region is already localized and the test prefix exercises that region. This abstraction removes confounding factors such as bug localization or test generation, focusing instead on the LLM's ability to complete a potentially fault-revealing test.

```
1 public void testCopyConstructor_noSideEffectOnAttributes()↩
      {
2     Safelist safelist1 = Safelist.none().addAttributes(↩
          TEST_TAG, TEST_ATTRIBUTE);
3     Safelist safelist2 = new Safelist(safelist1);
4     safelist1.addAttributes(TEST_TAG, "invalidAttribute");
5
6     \\ Original Oracle
          assertFalse(safelist2.isSafeAttribute(TEST_TAG,
          null, new Attribute("invalidAttribute",
          TEST_VALUE)));
7     \\ Generated Oracle
          assertFalse(safelist2.isSafeTag(TEST_TAG));
8 }
```

Fig. 4. Example of LLM-generated oracle Lacking Context

E. Further LLM Output Analysis

We manually examined a subset of eight test oracle generations from StarCoder and eight from GPT-4o to gain further insights into their behaviour.

For StarCoder, cases where oracles correctly identified failures in buggy code but incorrectly flagged correct code tended to fall into two main categories: over generation of oracles [6] and insufficient input context. Regarding over generation, StarCoder often produced more oracles than necessary. While this set might include the correct oracles, it also contains irrelevant ones that fail due to the limited context provided by the test prefix. For example, it was common to see oracles testing multiple unrelated functionalities. This limitation likely results from the narrow scope of the input prompt used in our experiments. Supplying broader context, such as the entire test class or suite, could help the model assign oracles to the correct test cases and improve their relevance. The second issue relates to the lack of context in the input itself. When only the test prefix is provided, StarCoder lacks sufficient information to generate reliable oracles. Without enough details about the code's purpose and usage, the model struggles to create representative oracles. Figure 4 shows a StarCoder-generated oracle failing to compile due to a missing `isSafe` method.

GPT-4o, often failed correct code for similar reasons to StarCoder, but also frequently overcomplicated comparisons. Figure 5 is an example where instead of using `assertEquals` to compare two values, it would use `assertArrayEquals`. This likely stems from GPT-4o trying to compare entire objects rather than just method results, aiming for content-based comparison. However, this can leads to both false positive and false negative results as they are comparing objects which were not originally intended to be compared as a part of the particular test case. For instance, the constructor and overridden `equals` method of the objects could mean the two instances of the object being compared are considered equal despite the values the test actually wants to compare being different, thus potentially resulting in a false positive. Conversely, if the `equals` method for the objects being compared has not been overridden or uses object references to compare, the two instances being compared (if they are different instances) will be returned as not equal, thus resulting in a false negative.

Both StarCoder and GPT-4o struggled with generating oracles that failed to compile, mainly due to missing context and

```
1 public void testLazilyParsedNumberDeserialization() {
2     LazilyParsedNumber expected = new LazilyParsedNumber("↩
          1.5");
3     LazilyParsedNumber actual = gson.fromJson("1.5", ↩
          LazilyParsedNumber.class);
4
5     \\ Original Oracle
          assertEquals(expected, actual);
6     \\ Generated Oracle
          assertArrayEquals(new LazilyParsedNumber[]expected,
          new LazilyParsedNumber[]actual);
7 }
```

Fig. 5. Example of overcomplicated LLM-Generated Oracle

over generation of code. When given only the test prefix, the models lacked details about methods, arguments, or expected outputs, leading to guesswork and compilation errors. Including the MUT helped but didn't fully resolve the issue, as it provided context for only one method. Without including the CUT, aligning output with the full test scenario remained difficult. Over generation of code was especially the case for Chain-of-Thought and Tree-of-Thought prompts, where the models often added unnecessary test logic.

To address these limitations, future work could explore providing more comprehensive prompts, such as including the test class or suite, to give LLMs a clearer understanding of the broader context. This might enable the models to allocate oracles to the appropriate test cases and reduce over generation of irrelevant test logic. Additionally, ensuring that prompts guide LLMs toward simpler, more direct assertions could help avoid overcomplicated oracle logic and improve both the accuracy and reliability of generated test oracles.

V. Threats to Validity

Data leakage. To reduce the risk of data leakage, we use the GHRB dataset [18], which includes bugs reported after StarCoder and GPT-4o training cutoff dates, ensuring the models have not seen these faults.

Sample size and generalisation. We used 36 randomly selected bugs from GHRB. The nature of these bugs may affect our findings, and results may not generalise beyond this sample.

Bug-revealing test case prefixes. Our study uses test case prefixes known to expose bugs. This limits our ability to assess whether LLMs can generate effective oracles without such prefixes or in correct code scenarios.

Lack of false positives analysis. We evaluate only buggy scenarios. However, in practice, most code is correct. It is also important to check whether LLMs generate false positives: assertions that wrongly fail on correct code [40], [41]. Evaluating oracle generation for bug-free units is an important direction for future work.

Randomness in LLM outputs. To account for the non-deterministic nature of LLMs [42], we ran prompts with each parameter combination five times. Our results indicate that the variance across these runs was low. Additionally, we used fixed values for temperature and top-p parameters. We acknowledge that exploring a broader range of values for these parameters may affect results and consider this an avenue for future work.

Prompt configuration scope. A potential threat to validity lies in the limited range of input prompt configurations explored. We only evaluated three scenarios (test prefix alone, test prefix with MUT, and test prefix with CUT), while excluding other possible variations such as removing comments, including dependencies, or using only method and class signatures. These alternative configurations may influence LLM behaviour and the resulting oracle generation in ways we did not capture. As such, our findings may not fully generalise across different prompt structures.

Scope of LLM selection. Finally, our study evaluates only one representative LLM from each category—one code-specific (STARCODER) and one general-purpose (GPT-4O). This choice was made for feasibility reasons and because the GHRB dataset is specifically designed to address data leakage concerns related to these models. *Future studies should explore a broader range of LLMs to assess the generality of our findings*.

VI. RELATED WORK

Automated unit test and oracle generation is a research area that has gained significant attention [14], [43]. Tools like EVOSUITE [32] and RANDOOP [2] generate tests with regressions assertions, while recent methods use neural models to generate assertions from test prefixes [38], [44], [45], [40], [46], [47]. Due to space constraints, in this section we highlight only key studies on LLM-based test and oracle generation.

LLMs for Test Case/Suite Generation. Recent studies have explored LLMs for automated test generation, focusing on their effectiveness, prompt design, and limitations. Yang et al. [9] found that LLM-generated unit tests had lower compilation rates and coverage compared to EVOSUITE. Alshahwan et al. [48] introduced TESTGEN-LLM, deployed at Meta, which improved 11.5% of test cases, with 73% accepted in production. Siddiq et al. [35] analyzed CODEX, GPT-3.5, and STARCODER, highlighting the impact of code context. Schäfer et al. [34] evaluated the LLM-driven test tool TESTPILOT, in terms of code coverage and failure detection. Lops et al. [36] introduced AGONETEST, noting low compilation rates and the influence of prompting strategies. Ouédraogo et al. [49] found that structured prompts improved test quality but struggled with complex code.

All of the mentioned studies focus on generating full test cases using LLMs. In contrast, our study addresses a more specific challenge: automated test oracle generation. This focus is well justified, as the oracle problem is one of the main bottlenecks in achieving full test automation [5]. By isolating test oracle generation, we remove confounding factors related to the test prefix (e.g., coverage, code quality) and focus solely on the test oracle's fault detection effectiveness.

LLMs for Test Oracle Generation Research related to test oracle generation with LLMs has advanced significantly, with various approaches exploring the potential of LLMs to automate and improve software testing processes. The work by Molina et al. [50] presents a roadmap for future research on the usage of LLMs for test oracle automation. Hossain et al. [13] introduced TOGLL, a method that leverages LLMs for generating test assertions while relying on the EVOSUITE tool for test prefix generation. The authors evaluate six different levels of contextual information. Hayet et al. [15] introduce CHATASSERT, an LLM-based test oracle generation tool with two modes of execution: generation and repair. In generation mode, it uses ChatGPT with a fixed prompt that includes summaries of the methods used in the test prefix, as well as similar examples in the form of other tests from the same test file. Konstantinou et al. [4] studied whether LLMs generate test oracles that focus on the implemented behaviour of the code or whether they are capable of producing non-regression oracles that capture the expected behaviour. The authors reuse the best-performing prompts from previous works such as TOGLL [13] and CHATTESTER [16]. Zhang et al. [29] investigated the performance of LLM-based assertion generation in terms of bug detection. The prompts employed by the authors contain the test prefix and MUT.

Our study is the first of its kind, substantially differing from previous works. First, most prior studies evaluate a single prompt type, varying only the fixed information about the MUT or CUT. The exception is Hossain et al., who explored multiple prompts with different levels of MUT context. However, no study has examined the combination of both input and prompt strategies. Second, we are the first to use the GHRB dataset—designed to reduce data leakage—for oracle generation, addressing a key limitation in earlier Java-based studies.

VII. CONCLUSIONS AND FUTURE WORK

This study provides a baseline for LLM-driven fault-revealing test oracle generation, an essential step toward reliable Prompt-ware. We evaluated two LLMs (STARCODER and GPT-4O) across four prompting techniques (zero-shot, few-shot, CoT, ToT) and four input contexts (test prefix, method under test, full class). Our large-scale evaluation (over 2,000 runs) highlights key challenges and insights that can help improve oracle generation in the FM era. Assertion compilability remains a major obstacle, and including full class context improves oracle quality by 12.9%. Simpler prompting techniques outperform CoT and ToT by 23%, and STARCODER consistently outperforms GPT-4O. Prompting technique emerges as the most impactful factor.

Future work should focus on improving assertion compilability, possibly through static analysis or LLM-driven refinement prompts [17]. Additionally, hybrid prompting strategies that combine structured reasoning (e.g., CoT) with simpler approaches (e.g., zero-shot or few-shot) may improve both accuracy and reliability.

ACKNOWLEDGMENTS

This work has been supported by the ITEA grant GENIUS (project number 23026).

REFERENCES

[1] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *2011 11th International Conference on Quality Software*. IEEE, 2011, pp. 31–40.

[2] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[3] S. Ruberto, J. Perera, G. Jahangirova, and V. Terragni, "From implemented to expected behaviors: Leveraging regression oracles for non-regression fault detection using llms," in *IEEE Conference on Software Testing, Verification and Validation Workshop*. IEEE, 2025, pp. 37–40.

[4] M. Konstantinou, R. Degiovanni, and M. Papadakis, "Do llms generate test oracles that capture the actual or the expected program behaviour?" *arXiv e-prints*, pp. arXiv–2410, 2024.

[5] 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.

[6] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 1178–1189.

[7] V. Terragni, A. Vella, P. Roop, and K. Blincoe, "The future of ai-driven software engineering," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025.

[8] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[9] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang *et al.*, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1607–1619.

[10] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[11] S. B. Hossain, R. Taylor, and M. Dwyer, "Doc2oracll: Investigating the impact of documentation on llm-based test oracle generation," *Proc. ACM Softw. Eng.*, Jun. 2025.

[12] S. M. Khandaker, F. Kifetew, D. Prandi, and A. Susi, "Augmentest: Enhancing tests with llm-driven oracles," in *IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2025.

[13] S. Binta Hossain and M. Dwyer, "Togll: Correct and strong test oracle generation with llms," *arXiv e-prints*, pp. arXiv–2405, 2024.

[14] G. Jahangirova and V. Terragni, "Sbft tool competition 2023 - java test case generation track," 2023, pp. 61–64.

[15] I. Hayet, A. Scott, and M. d'Amorim, "Chatassert: Llm-based test oracle generation with external tools assistance," *IEEE Transactions on Software Engineering*, 2024.

[16] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[17] R. Ravi, D. Bradshaw, S. Ruberto, G. Jahangirova, and V. Terragni, "LLM-LOOP: Improving LLM-Generated Code and Tests through Automated Iterative Feedback Loops," in *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2025.

[18] J. Y. Lee, S. Kang, J. Yoon, and S. Yoo, "The github recent bugs dataset for evaluating llm-based debugging applications," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024, pp. 442–444.

[19] A. Bodicoat, V. Terragni, and G. Jahangirova, "Replication Package for: Understanding LLM-Driven Test Oracle Generation - AIware 2025," https://doi.org/10.6084/m9.figshare.30472256, 2025, accessed: 2025-10-30.

[20] B. Kitchenham and S. L. Pfleeger, "Principles of survey research: part 5: populations and samples," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 5, pp. 17–20, 2002.

[21] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284–292.

[22] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[23] 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.

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[25] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[26] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[27] Vellum AI, "Tree of thought prompting: What it is and how to use it," 2023. [Online]. Available: https://www.vellum.ai/blog/tree-of-thought-prompting-framework-examples

[28] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[29] Q. Zhang, W. Sun, C. Fang, B. Yu, H. Li, M. Yan, J. Zhou, and Z. Chen, "Exploring automated assertion generation via large language models," *ACM Transactions on Software Engineering and Methodology*.

[30] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[31] OpenAI, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[32] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[33] H. Face, "Starcoder model card," 2025, accessed: 2025-01-27. [Online]. Available: https://huggingface.co/bigcode/starcoder

[34] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[35] M. L. Siddiq, R. H. Tanvir, and N. Ulfat, "Exploring the effectiveness of large language models in generating unit tests."

[36] A. Lops, F. Narducci, A. Ragone, M. Trizio, and C. Bartolini, "A system for automated unit test generation using large language models and assessment of generated test suites," *arXiv preprint arXiv:2408.07846*, 2024.

[37] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[38] 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.

[39] W. Xiong, Y. Guo, and H. Chen, "The program testing ability of large language models for code," *arXiv preprint arXiv:2310.05727*, 2023.

[40] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2130–2141.

[41] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards more realistic evaluation for neural test oracle generation," 2023. [Online]. Available: https://arxiv.org/abs/2305.17047

[42] S. Cho, S. Ruberto, and V. Terragni, "Metamorphic testing of large language models for natural language processing," in *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2025.

[43] V. Terragni and M. Pezzè, "Effectiveness and challenges in generating concurrent tests for thread-safe classes," in *33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 64–75.

[44] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated assertion generation via information retrieval and its integration with deep learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 163–174.

[45] 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*, 2022, pp. 54–64.

[46] S. B. Hossain, A. Filieri, M. B. Dwyer, S. Elbaum, and W. Visser, "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*, 2023, pp. 120–132.

[47] J. Shin, H. Hemmati, M. Wei, and S. Wang, "Assessing evaluation metrics for neural test oracle generation," *IEEE Transactions on Software Engineering*, 2024.

[48] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 185–196.

[49] W. C. Ouedraogo, K. Kabore, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyande, "Llms and prompting for unit test generation: A large-scale evaluation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2464–2465.

[50] F. Molina, A. Gorla, and M. d'Amorim, "Test oracle automation in the era of llms," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–24, 2025.