

TOGLL: Correct and Strong Test Oracle Generation with LLMs

Soneya Binta Hossain
Department of Computer Science
University of Virginia
sh7hv@virginia.edu

Matthew B. Dwyer
Department of Computer Science
University of Virginia
matthewbdwyer@virginia.edu

Abstract—Test oracles play a crucial role in software testing, enabling effective bug detection. Despite initial promise, neural methods for automated test oracle generation often result in a large number of false positives and weaker test oracles. While LLMs have shown impressive effectiveness in various software engineering tasks, including code generation, test case creation, and bug fixing, there remains a notable absence of large-scale studies exploring their effectiveness in test oracle generation. The question of whether LLMs can address the challenges in effective oracle generation is both compelling and requires thorough investigation.

In this research, we present the first comprehensive study to investigate the capabilities of LLMs in generating correct, diverse, and strong test oracles capable of effectively identifying a large number of unique bugs. To this end, we fine-tuned seven code LLMs using six distinct prompts on a large dataset consisting of 110 Java projects. Utilizing the most effective fine-tuned LLM and prompt pair, we introduce TOGLL, a novel LLM-based method for test oracle generation. To investigate the generalizability of TOGLL, we conduct studies on 25 unseen large-scale Java projects. Besides assessing the correctness, we also assess the diversity and strength of the generated oracles. We compare the results against EvoSuite and the state-of-the-art neural method, TOGA. Our findings reveal that TOGLL can produce 3.8 times more correct assertion oracles and 4.9 times more exception oracles than TOGA. Regarding bug detection effectiveness, TOGLL can detect 1,023 unique mutants that EvoSuite cannot, which is ten times more than what TOGA can detect. Additionally, TOGLL significantly outperforms TOGA in detecting real bugs from the Defects4J dataset.

I. INTRODUCTION

Software dominates almost every aspect of our lives, including safety-critical domains such as healthcare and autonomous transportation. Even seemingly minor software bugs can cause large-scale system outages, security breaches, and loss of lives [1]–[3]. Thus, ensuring software reliability through rigorous testing and bug detection is paramount. Test oracles, the foundation of effective testing, play a pivotal role in the early detection of software bugs [4]–[7].

Typically, a test suite consists of a set of test cases, where each test is composed of a *test prefix* that exercises a certain part of a program, and a *test oracle* verifies whether the executed behavior matches the expected behavior [8]. Test oracles can be categorized into two main types: *assertion oracles*, which judge the correctness of the output state of the program, and *exception oracles*, which judge whether erroneous input states are detected by the SUT.

The efficacy of test oracles is determined by their ability to detect bugs. For effective bug detection, test oracles should be *correct* and *strong*. A test oracle is considered correct if it aligns with the expected program behavior, thereby avoiding false alarms. Additionally, a test oracle is deemed strong if it can detect deviations from intended program behavior. Just because a test oracle is correct does not mean it is strong, as demonstrated in [5].

Researchers have investigated the application of natural language processing (NLP) and pattern matching techniques for generating test oracles from code comments and natural language documentation [9]–[13]. These approaches can generate both assertion oracles, which verify that a program’s actual output matches its expected output, and exception oracle, which capture the program’s anticipated exceptional behaviors during testing. Recent advancements have seen the application of neural techniques for generating test oracles, utilizing transformer-based models that learn from the method under test (MUT) and developer-written test cases [14]–[16]. Building upon these methods, the development of TOGA [17]—a neural-based test oracle generator—outperformed prior work in detecting real faults. However, a recent study shows that this state-of-the-art neural-based test oracle generation method, when given a test prefix, the MUT and its docstring, generates assertion oracles only 38% of the time. Moreover, when it does generate assertion oracles 47% are false positives, and for exception oracles the false positive rate is 81%. Despite significant research focused on generating test oracles, automated generation of correct and strong test oracles has proven challenging and the state of the art suffers from: generalizability issues, high false positive rates and limited bug detection effectiveness [5]. These challenges emphasize the ongoing need for further investigation, marking it as a significant and open research problem.

Large language models (LLMs) have shown promise in automating diverse software engineering tasks such as code completion [18], program comprehension [19], automated repair [20], and test generation [21]–[23]. Most prior work on ML-based test generation has focused on the generation of test prefixes – inputs to the system under test — with the goal of yielding substantial test coverage. CodaMosa [22] uses LLMs to escape coverage plateaus in search based test input generation and find it can improve coverage on small to

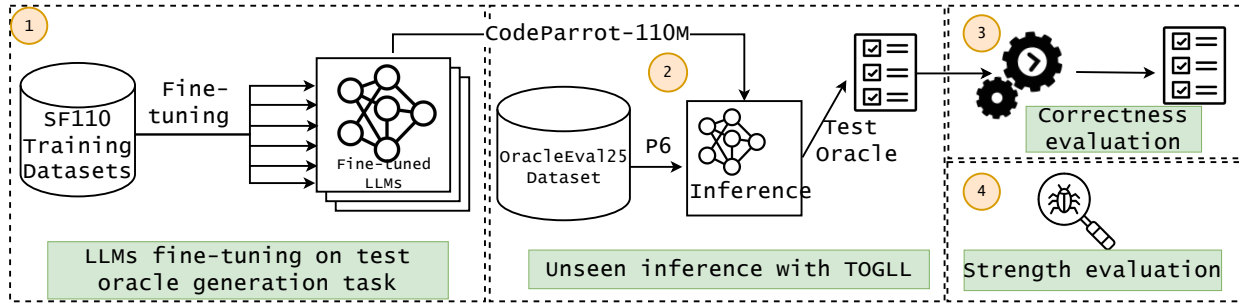


Fig. 1. A general overview of our research approach to investigating LLM-based automated oracle generation.

moderate Python programs. A recent attempt to use LLMs to generate test prefixes for large-scale Java programs was not very successful [23]; it achieved only 2% statement coverage on large-scale Java projects compared to over 90% for EvoSuite’s [24] search-based approach. The LLM-based TESTPILOT [21] approach generates both test prefixes and test oracles for JavaScript. It produces better coverage than state-of-the-art search-based JavaScript test generation technique on a variety of small programs. With regard to test oracles, TESTPILOT was shown to generate significant numbers of non-trivial assertions that are distinct from those generated by other methods, but this work is limited to JavaScript and did not evaluate oracles correctness, diversity and strength aspects thoroughly.

In this paper, rather than task the LLM with generating both test prefix and test oracle, as TESTPILOT did, we decouple these problems. We utilize state-of-the-art test generation methods, such as EvoSuite, to create test prefixes and concentrate on leveraging LLMs for test oracle generation. Initially, we fine-tune seven code LLMs, with sizes ranging from 110M to 2.7B parameters, employing six prompt formats that vary in the information they incorporate about the method under test (MUT), its documentation, and the test prefix. Employing the most effective model and prompt pair, we introduce our LLM-based method, TOGLL.

It is important to note that TOGLL generated oracles are not regression oracles. Regression oracles are generated based on the executed program behavior. For example, EvoSuite executes the code, collects all actual results values, knows the types, and then generates oracles based on those. This is why regression oracles are not capable of detecting bugs in current program versions. TOGLL, in contrast, does not execute the code and is not aware of the test execution results. It uses Javadoc documentation, MUT/MUT signature and test prefix to generate oracles. Some of the prompts (see Table I) do not even use MUT (P1, P2) at all and some (P3, P4) only use MUT signature, not the entire implementation.

To evaluate TOGLL’s generalizability on unseen data, we conduct a comprehensive study on 25 real-world large-scale Java projects and compare the correctness of the generated oracles with those produced by the SOTA neural method, TOGA [17] and EvoSuite. Subsequently, we assess the strength of the generated oracles through a large-scale mutant detection study and a real bug detection study, highlighting the

unique advantages of TOGLL-generated oracles over alternative methods. To the best of our knowledge, this represents the first extensive study to explore the application of LLMs in test oracle generation, examining seven code LLMs, six prompts, and three large datasets, while evaluating multiple aspects of the generated oracles, including correctness, diversity, and bug detection effectiveness.

In summary, the main contributions of the paper lie in:

- Constructing a large dataset from 110 large-scale Java projects to fine-tune code LLMs, aiming to investigate their performance in generating effective test oracles.
- Evaluating the generalizability of fine-tuned LLMs in generating effective test oracles on 25 unseen projects and compare them with SOTA methods.
- Evaluating the impact of different contextual information on oracle generation accuracy through six specially designed prompts.
- Demonstrating that well-tuned and prompted LLMs can generate correct, strong, and diverse test oracles, with much lower false positive rates than SOTA-methods and can detect large number of unique bugs neither detected by EvoSuite nor SOTA neural method.
- Releasing all datasets, fine-tuned models, and code to enable replication of our study and the community to build on the work.

This work offers a promising starting point for LLM-based test oracle generation, identifies directions for future improvement of such methods, and thereby has the potential to lead to impactful new capabilities for software engineers.

II. APPROACH

Figure 1 presents an overview of our research approach. Step 1 of Figure 1 depicts fine-tuning a collection of LLMs with different prompts for test oracle generation task; we discuss this in Section II-A. Step 2 of Figure 1 selects the best performing fine-tuned model and prompt from Step 1 to define TOGLL and perform oracle inference on the unseen data. Step 3 assesses the correctness of TOGLL generated oracles through test execution; we discuss this in Section II-B1. Finally, step 4 assesses the strength of TOGLL-generated oracles using mutation testing and real bug detection from Defects4J; we discuss this in Section III-D and III-E.

A. Supervised Fine-tuning

This section covers the datasets, prompts, and LLMs that are fine-tuned for the test oracle generation task.

1) *SF110 Dataset*: We construct this dataset using the SF110 EvoSuite benchmark [25], which comprises 110 open-source Java projects with over 23,000 Java classes sourced from the SourceForge repository. This benchmark is widely recognized for unit test generation. To construct our dataset, we utilize the test cases generated by EvoSuite for each project. Given that a single test case can contain multiple assertion oracles or even a combination of assertion and exception oracles, we process each test case. This involves decomposing them into individual test cases, ensuring that each contains only a single assertion or exception oracle. For each decomposed test case, we extracted the method under test (MUT) along with its associated documentation to prepare the dataset. The dataset is comprised of tuples $((p_i, m_i, d_i), o_i)$, where p_i is the test prefix, m_i is the MUT, and d_i is the docstring (if available) and o_i is the ground truth test oracle, either assertion or exception. This dataset is used to fine-tune the LLMs and has been partitioned into three distinct subsets: 90% for training, 5% for validation, and another 5% for testing.

2) *Prompt Designing*: Even though LLMs are powerful, they are not specifically designed and pre-trained for test oracle generation. Thus, fine-tuning with effective prompts is necessary for optimal performance. We have designed six different prompts, each tailored to a specific use case scenario.

- Prompt 1 (**P1**) consists of only the test prefix ('prefix').
- Prompt 2 (**P2**) additionally includes the MUT documentation strings ('prefix + [sep] + doc'). The '[sep]' separator token is different based on different LLMs and their respective tokenizer.
- Prompt 3 (**P3**) instead of docstrings, includes the MUT signature ('prefix + [sep] + mutsig').
- Prompt 4 (**P4**) includes both MUT signature and docstrings in addition to prefix ('prefix + [sep] + doc + [sep] + mutsig').
- Prompt 5 (**P5**) includes the code for the entire MUT ('prefix + [sep] + mut').
- Prompt 6 (**P6**) adds the docstring to P5 ('prefix + [sep] + doc + [sep] + mut').

These prompts are designed to cover diverse real use case scenarios. For example, when documents are not available, P1, P3, and P5 can be used. When the MUT implementation is not available, P1 - P4 can be used, and when it is available, P5 or P6 can be used. Designing these prompts allows us to assess their relative accuracies and the impact of different information on the oracle generation performance.

3) *Code LLMs*: Decoder-only LLMs are most suitable for code generation tasks [26]. Smaller models are cost-effective, produce a lower carbon footprint [27], and can be more easily fine-tuned with limited GPU resources. Therefore, in our study, we prefer decoder-only LLMs that are pre-trained on various code generation tasks within the software development life

cycle (SDLC) [26], [28], smaller in size, publicly available on Hugging Face [29], and well-documented for fine-tuning. We chose seven LLMs because they are pre-trained on multiple programming languages while meeting all the above criteria.

CodeGPT [30]: It is a GPT-style language model with 110M parameters. We choose to fine-tune this model for two main reasons. Firstly, both our training and unseen inference datasets came from Java projects, and this model is also pre-trained on Java programming language. Secondly, this model is pre-trained on the code completion task, which aligns with our designed prompts. We fine-tune the microsoft/CodeGPT-small-java model from Hugging Face.

CodeParrot [31]: In our study, we fine-tune the 110M parameters model, which was pre-trained for the code generation task on nine different languages: Java, JavaScript, PHP, Python, C#, C++, GO, Ruby, and TypeScript. We have fine-tuned the codeparrot/codeparrot-small-multi model from Hugging Face. Our experimental study suggests that despite its small size, the fine-tuned CodeParrot model generalizes strongly on unseen data and can outperform larger models with billions of parameters.

CodeGen [32]: CodeGen is a family of autoregressive language models with four different trainable parameter sizes: 350M, 2B, 6B, 16B. In this paper, we have fine-tuned 350M and 2B Multi variant models that are pre-trained on dataset that encompasses six programming languages: C, C++, Go, Java, JavaScript, and Python. Because of their great capability in comprehending and generating codes, we fine-tune them for test oracle synthesis task.

PolyCoder [33]: PolyCoder is a family of large language models with three trainable parameter sizes: .4B, 2.7B, and 16B. This model was trained on 249 GB of code across 12 programming languages. In this paper we have fine-tuned .4B and 2.7B model for the test oracle generation task.

Phi-1 [34]: Phi-1 is a transformer language model with 1.3 billion parameters, initially pre-trained on Python code from a variety of data sources for the task of Python code generation. Despite its primary training on Python, our study observed that it performs on par with other models when fine-tuned on Java code for generating test oracles.

B. Unseen Inference

1) *OracleEval25 Dataset*: To investigate the generalizability of the fine-tuned models, we utilize the dataset described in [5], comprising 25 large-scale industrial standard Java projects. Of these, 17 originate from the Apache Commons Proper [35], while the remainder are sourced from GitHub. These latter projects are characterized by large codebases with multiple modules, with up to 10k active users per project, and are frequently employed in software engineering research for the evaluation of test cases and test oracles [6], [7], [36]. Given their broad coverage of 21 domains and significant variation in developer metrics, the 25 artifacts selected for this study provide a robust framework for assessing fine-tuned model's ability to generalize across a spectrum of real-world

projects. Overall, the dataset consists of 271K source lines of code (SLOC), 331K lines of JavaDoc, 1214K test suite SLOC and 223.5K test cases. Similar to the SF110 dataset, each input sample of this dataset contains the test prefix, MUT and docstring. We refer to this dataset as OracleEval25.

2) *TOGLL*: From the fine-tuning results, we identify CodeGen-350M and CodeParrot-110M as the top two models due to their high accuracy. We also select the three best-performing prompts: P4, P5, and P6. As different fine-tuned models may generalize differently to unseen data, and due to the differences between datasets, different prompts may work better. For this reason, we conduct a small-scale study, performing unseen inference with these two models and three prompts on projects from the OracleEval25 dataset. Our results suggest that even though CodeGen-350M is three times larger than CodeParrot-110M, CodeParrot-110M fine-tuned with P6 performs better on the unseen project than CodeGen on several metrics: total input processed, compilation error, and false positive count. Furthermore, the inference time with CodeGen-350M is longer than CodeParrot. Consequently, we define TOGLL as a fine-tuned CodeParrot-110M operating on P6. We perform rest of the study with TOGLL on the unseen OracleEval25 dataset II-B1.

3) *Baseline*: TOGA is a neural method for both assertion and exception oracle generation [17]. TOGA utilizes CodeBERT [37] model as backbone and fine-tuned for the oracle generation task. TOGA takes the test prefix generated by EvoSuite, its associated MUT and docstring (if available) and generates either an assertion or an exception oracle. An example of EvoSuite generated test cases with assertion and exception oracle is shown in Figure 2. The prefix part is marked with yellow color. The first test prefix involves pushing the number 10 onto the stack, followed by popping this value from the stack. For this scenario, an assertion oracle is anticipated to validate that the value retrieved is indeed 10. The second test attempts to pop from an empty stack, expecting the MUT to throw an exception. If no exception is thrown, the test should fail.

We selected TOGA as our baseline method for several reasons: 1) TOGA represents the SOTA neural method, having surpassed specification, search, and neural-based techniques by detecting 57 bugs in Defects4J [36]; 2) TOGA utilizes the same set of information as P6, which is our most comprehensive prompt, thereby providing a fair basis for comparison; 3) TOGA has been previously evaluated on the OracleEval25 dataset, which is also employed in our study, ensuring consistency in our comparative analysis [5].

4) *Metrics*: With TOGLL, we generate test oracles for all projects from the OracleEval25 dataset. This dataset framework includes all necessary artifacts for executing the generated test oracles when integrated with the respective test prefixes. Consequently, we integrate the generated oracles into the test cases and execute the complete test suites. We calculate accuracy as the success rate, which denotes the percentage of non-empty test oracles that were successfully passed during test execution, indicating their alignment with

```

Test Prefix With Assertion Oracle
public void testPushAndPopStack() {
    Stack<Integer> stack = new Stack<>();
    stack.push(10);
    Integer val = stack.pop();
    assertEquals(Integer.valueOf(10), val);
}

Test Prefix With Exception Oracle
public void testPopEmptyStack() {
    Stack<Integer> stack = new Stack<>();
    try {
        stack.pop();
        fail("");
    } catch (EmptyStackException e) {
        // Test passed
    }
}

```

Fig. 2. Test cases with assertion and exception oracles. The test prefix part is marked with yellow color.

expected program behavior. We compute the success rate using Equation 1

$$\text{SuccessRate}(P) = \frac{T - (T_{ce} + T_{fp} + T_{em})}{T} \quad (1)$$

where T is the number of total test prefixes in project P , T_{ce} is the number of compilation errors, T_{fp} is the number false positive tests, and T_{em} the number of empty oracles. We compute the SuccessRate metric for both assertion and exception oracles predicted by TOGLL and compare them to those of our baseline, TOGA.

To assess the bug detection effectiveness of the TOGLL-generated oracles, we perform both mutation testing and real bug detection study on Defects4J. Mutation testing introduces small changes, or mutations, to the source code to create numerous slightly altered versions of the software, called *mutants* [38], [39]. Studies showed that there is a statistically significant correlation between mutant detection and real fault detection [40]–[42]. A test suite’s ability to detect these mutants can be a direct measure of its bug detection effectiveness [6], [7]. Therefore, we compute the mutant killing score as a metric to assess and compare the bug detection effectiveness of TOGLL-generated oracles with both EvoSuite and TOGA. Additionally, we also record unique mutants killing score which indicates the unique capability of each oracle generation method. Besides the mutation study, we also perform real bug detection study on the Defects4J dataset and compare the results with the TOGA baseline.

III. EXPERIMENTAL STUDY

To assess the efficacy of LLM-generated test oracles, this study investigates several key areas: effectiveness of different code LLMs in generating test oracles, the impact of different prompts on the oracle generation accuracy, the ability of fine-tuned models to generalize to unseen data, comparative accuracy among different oracle types, diversity of the generated oracles and the overall efficacy of these oracles in detecting bugs. To this end, we answer the following five research questions.

TABLE I
TEST ORACLE GENERATION PERFORMANCE OF LLMs ON DIFFERENT PROMPTS. TOP TWO ACCURACIES FOR EACH PROMPT ARE SHOWN IN **BOLD**. THE LAST ROW SHOWS PROMPT-WISE AVERAGE.

Code LLM	P1	P2	P3	P4	P5	P6	Prompt Details
CodeGPT-110M	54.4	63.5	72.7	73	75	74.1	P1: prefix
CodeParrot-110M	56.4	65.6	76.1	76.2	77.7	77.7	P2: prefix + [sep] + doc.
CodeGen-350M	56.7	66.3	77.4	77.5	79.3	79	P3: prefix + [sep] + mutsig.
PolyCoder-4B	56.4	64.3	74.5	74.9	76.8	76.8	P4: prefix + [sep] + doc. + [sep] + mutsig
Phi-1(1.3B)	53.42	61.5	75.5	73.4	77.6	74.3	P5: prefix + [sep] + mut
CodeGen-2B	55.7	65.5	75	74.7	76.5	76.8	P6: prefix + [sep] + doc. + [sep] + mut
PolyCoder-2.7B	55	64	74.8	74.7	76.6	76.9	
Avg:	55.4	64.4	75	75	77	76.5	

RQ1: What LLM and prompting approaches are effective for test oracle generation? *Increasing context in the prompt improves accuracy of oracle generation up to 79% for a range of LLMs, but smaller LLMs meet or exceed the performance of larger LLMs when fine tuned.*

RQ2: How well does TOGLL, i.e., fine-tuned model generates correct test oracles for unseen projects? *TOGLL generates up to 4.9 times the number of correct test oracles compared to prior neural based approach.*

RQ3: How diverse are LLM-generated assertions relative to those generated by EvoSuite? *TOGLL-generated oracles vary substantially in terms of the specific assert statements used as well as in the variables and expressions references in those oracles compared to those generated by EvoSuite.*

RQ4: How strong are TOGLL-generated assertions at identifying unique mutants? *TOGLL generated oracles kill nearly 10 times more unique mutants than prior SOTA method and fall only 17% short of EvoSuite’s bug detection effectiveness.*

RQ5: How strong are TOGLL-generated oracles at detecting real bugs from Defects4J? *TOGLL-generated exception and explicit assertion oracles detected 106% more bugs compared to the prior SOTA method TOGA.*

A. RQ1: Selecting LLMs and Prompts for Oracle Generation

In this research question, we fine-tune various large language models, as outlined in Section II-A3, utilizing six unique prompts discussed in Section II-A2. The purpose of this study is to determine the most effective combination of model and prompt format for the test oracle generation task. This approach allows us to understand the impact of prompts on model efficacy and guides the selection of the most effective model-prompt pair for a deeper investigation.

1) *Experimental Setup:* For this study, we utilized the SF110 training dataset discussed in Section II-A1, which includes 159,073 input samples. Each sample contains a unique identifier, the method under test (MUT), a test prefix, and a docstring, from which we constructed six prompts as outlined

in Section II-A2. For the fine-tuning process of the code LLMs, we employed a GPU setup consisting of 4 A100 GPUs, each with 40GB of memory. To facilitate parallel processing and accommodate larger models such as CodeGen-2B and PolyCoder-2.7B within the 40GB memory limit, we utilized PyTorch Accelerator and DeepSpeed [43]. The models were fine-tuned using a learning rate of $2.5e^{-5}$ and the AdamW optimizer. Each LLM was fine-tuned across all six prompts, and accuracy was computed on the validation dataset using the ‘exact match’ metric. This metric may underestimate the overall accuracy, as a generated oracle may not be an exact match but can still be correct, as illustrated in Figure 4.

2) *Results:* Table I displays the specifics of each prompt—detailing the information used in each prompt—on the right side. P1 utilizes only the test prefix as input, prompting the model to generate either an assertion or an exception oracle. P2 adds a docstring in addition to the test prefix, while P3 includes a method signature. P4 includes both docstring and method signature after the test prefix. P5 includes the entire MUT code with the test prefix, whereas, P6 contains both the docstring and entire MUT code with the test prefix. On the left side of Table I, we present the performance of seven fine-tuned code models in generating test oracles for all six prompts.

The average accuracy of the fine-tuned models on P1 is around 55.4%. If we add the available docstring in the prompt, the accuracy is increased by around 10%. Instead of adding the docstring, if we add the method signature to the prompt, the accuracy is increased by 20%, resulting in on average 75% accuracy for P3. P4 includes both docstring and method signature with test prefix, which only see a marginal improvement for few models. Most accuracy improvement was observed when we add the entire code of the MUT in P5 and P6, with P5 achieving highest accuracy of 77%. We also observe that adding more context can also decrease the accuracy, such as for P5 and P6. Overall, we see a consistent accuracy improvement as we go from prompt P1 to P5.

We observe that including method signature in the prompt works significantly better than including the docstring. Even when including both method signature and docstring – P4 – only marginal improvement was observed. This observation also true for P5 and P6, adding the docstring did not improve much accuracy when either method signature or entire method code is available. However, from prompt P1 to P2, accuracy is improved by 10%, indicating the value of docstrings. The quality of the docstrings is not considered in this study. This is a promising future direction as we believe that documentation that combines natural language description of requirements with references to API elements might yield better results.

Among the seven code models evaluated, CodeGen-350M emerged as the top performer, closely followed by CodeParrot-110M, both highlighted in bold. The three top performing prompts are P4, P5, P6. RQ2 is performed on the unseen OracleEval25 dataset. Our investigation shows that CodeParrot-110M generalizes better on the unseen data than CodeGen-350M. Also, although, P5 works better in this study we find that P6 works better on the OracleEval25 dataset, which has more Javadoc (for 75% samples) compared to the SF110 dataset (45%) and P6 can leverage those Javadoc to generate better test oracles.

RQ1 Findings: LLMs are capable of accuracy levels above 79% in generating test oracles. Prompts containing increased contextual information enable higher accuracy, but perhaps surprisingly larger LLMs when fine-tuned do not offer significant advantages.

B. RQ2: Assessing Test Oracles Correctness

RQ1 investigates various LLMs’ test oracle generation performance when they are fine-tuned on the training portion of the SF110 dataset and their accuracy is evaluated on the validation portion of the same dataset. However, a fine-tuned model may not generalize when applied to unseen data, a common challenge in LLM applicability [44]. To investigate the generalizability of our method TOGLL, we perform this study on the unseen OracleEval25 dataset discussed in Section II-B1. A test oracle is correct if it captures the expected behavior of a software system. Generated test oracles that do not compile, that disagree with software behavior, or that are vacuous are considered incorrect. We study the rate of success in generating correct oracles for both TOGLL and TOGA.

1) *Experimental Setup:* We generate test oracles for the OracleEval25 unseen dataset using our method TOGLL, which is a fine-tuned CodeParrot-110M model operating on P6, which includes the MUT, test prefix and docstrings. This dataset consist of a total of 202,434 assertion prefixes requiring TOGLL to generate an assertion oracle for each input sample and 21,123 exception prefixes requiring TOGLL to generate an exception oracle. We have used a single A100 GPU with 10GB of memory to perform the inference task. The dataset keeps a record of the metadata so that the generated test oracles can

be integrated with the test prefixes, enabling test validation to be performed later.

Unlike RQ1, which assessed accuracy using the ‘exact match’ metric, RQ2 involves executing the integrated test cases—combining the test prefix with the generated test oracles—to collect detailed metrics for each type of oracle. For both assertion and exception oracles, we compute success rates across all 25 projects using Equation 1, and compare these rates to those achieved by TOGA. Counting empty and non-compiling assertions is straightforward. For false positives, a failing test oracle can indicate either an incorrect oracle or a bug in the MUT. If the MUT is known to be correct, a failing test oracle signals a false alarm. We use this well-established technique for test oracle evaluation [5] to evaluate TOGLL-generated oracles. The OracleEval25 dataset includes the latest stable releases of projects with no known bugs. Therefore, any failure of a generated test oracle suggests it is incorrect.

2) *Results:* In Figure 3, we present and compare our LLM-based method, TOGLL’s correct test oracle generation performance with deep-learning-based method, TOGA. The Y-axis displays success rate as a percentage. The X-axis enumerates the names of 25 projects. Each project is represented by two sets of vertical bars: blue for TOGLL and red for TOGA. Solid blue bars show TOGLL’s success rate in generating assertion oracles, while blue bars with diagonal lines illustrate its success in generating exception oracles. Similarly, solid red bars represent TOGA’s assertion oracle success rates, and red bars with diagonal lines show its performance in generating exception oracles.

Figure 3 shows that TOGLL achieved significantly higher success rate across all projects for both assertion and exception oracle generation. For example, for apache commons-number, TOGA could achieve a success rate of only 1%, whereas, TOGLL achieve significantly higher accuracy of 74.7%. Similarly, for http-request, TOGA achieved 14% where as TOGLL achieved 83% success rate. Statistical analysis reveals a significant difference in the average success rates, with TOGLL achieving a mean of 63% compared to TOGA’s mean of 16.4%. A t-test yielded a t-statistic of 12 and a p-value of $1.1e^{-11}$, indicating a statistically significant difference in performance between the two methods, with TOGLL outperforming TOGA in terms of correct test oracle generation.

TOGA uses a non-ML method to generate five types of assertion candidates using predefined templates, and then an ML model ranks the candidates to find the best assertion. We found that around 62% of the time, the model could not generate any assertions. Additionally, TOGA uses the most frequent constants and variables to generate assertEquals oracles, resulting in high false positives. TOGLL is not restricted to certain types of assertions and does not use frequently appearing values to generate assertions. Moreover, TOGLL makes more effective use of Javadoc than TOGA, which showed almost no improvement with Javadoc. TOGLL’s training data and model backbone could also contribute to its superior performance, but this requires further investigation.

For exception oracle, the performance difference is even

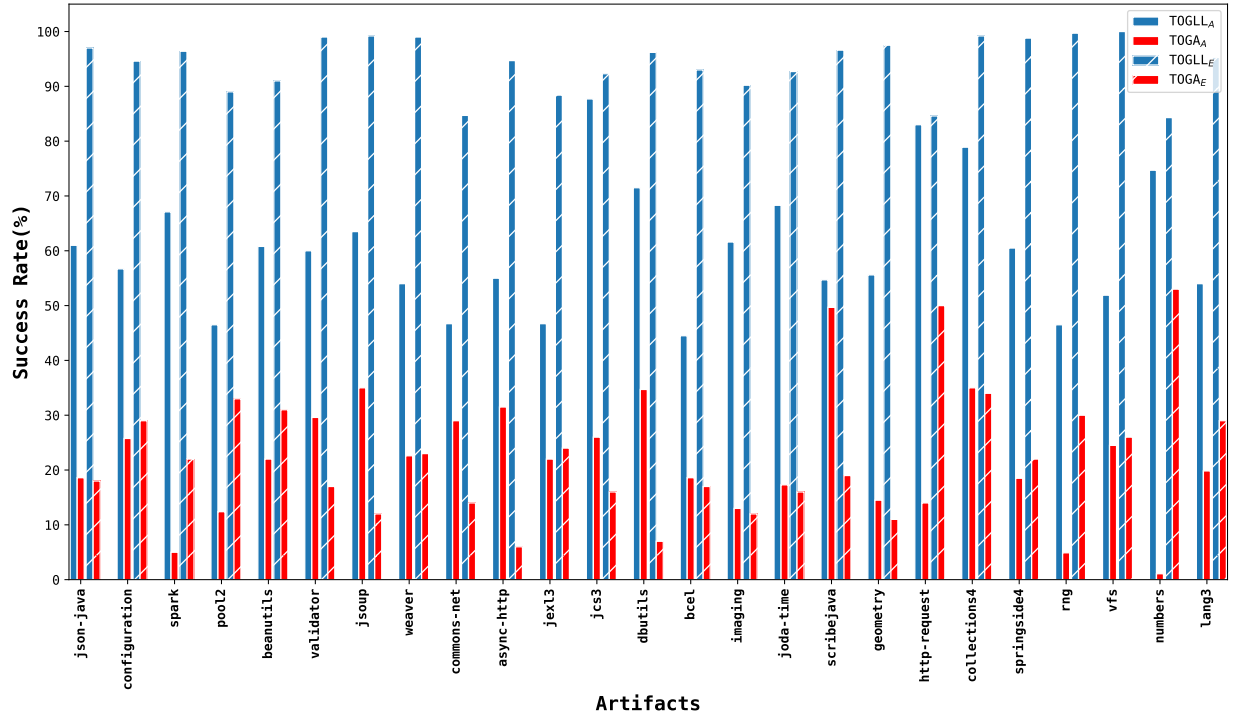


Fig. 3. Comparison of correct test oracle generation performance of TOGLL vs. TOGA

higher. TOGA’s average success rate is 18.86%, while TOGLL achieves a significantly higher average of 93.4%. A Mann-Whitney U test further confirms a statistically significant performance difference between TOGLL and TOGA, highlighting TOGLL’s superior performance in generating exception oracles compared to TOGA ($U = 625.0$, $p = 1.4e^{-09}$). Javadoc typically specifies exception conditions for a MUT and TOGLL’s ability to leverage these Javadoc and the specified exception conditions is a key factor contributing to its superior performance.

RQ2 Findings: TOGLL generates significantly more correct test oracles than TOGA; bettering it by 3.8 times and 4.9 times for assertion oracles and exception oracles, respectively.

C. RQ3: Investigating Diversity in Oracle Generation

Since TOGLL is fine-tuned on EvoSuite-generated test oracles, one concern is whether TOGLL simply learns to generate EvoSuite oracles. To this end, RQ3 investigates the diversity of the generated test assertions. More specifically, we investigate whether during unseen inference TOGLL follows the assertion distribution observed during training and also to what extent the generated assertions exactly match the ground truth assertions. These metrics can provide insights into the diversity of the generated assertions. Diversity is an important aspect of test oracle generation, allowing TOGLL to complement developer written oracles or those generated by other tools and thereby enable detection of faults that would

otherwise be undetected. In RQ4 and RQ5, we investigate the ability of TOGLL to detect unique faults.

1) *Experimental Setup:* To set up this study, we utilize the training dataset and results from unseen inference in RQ2. We divided the assertions into seven categories, consisting of the most frequently used JUnit assertions investigated in previous studies [5], [7], [17]. We count the total number of assertions in each category and calculate the distribution relative to the total number of assertions in the training data. We follow a similar process to compute the categorical distribution of TOGLL generated assertions during unseen inference. Furthermore, we compute the percentage of TOGLL generated assertions that are an exact match for ground truth assertions in our dataset.

2) *Results:* Table II, Column 1 lists the assertion categories, Column 2 shows the distribution of different types of assertions within the training dataset. Column 3 presents the breakdown of the oracles generated by TOGLL for the unseen dataset. Column 4 presents the percentage of exact match assertions for each category, reflecting the instances where the TOGLL-generated assertions exactly match the ground truth assertions.

There is a clear evidence that the distribution of assertions generated by TOGLL is quite different from the training data. The training data has a sharp peak for `assertEquals`, whereas the inference data has lower peaks for two categories. Moreover the tails of the distributions have a much higher share of samples indicating that generation of assertions is much less concentrated than training data. With regard to assertion category TOGLL is more diverse in generating assertion oracles.

We see that out of the 194k generated assertions, only

TABLE II

DISTRIBUTION OF ASSERTION ORACLES ACROSS DIFFERENT CATEGORIES

Assertion Category	Assertion Distribution		
	Training	Inference	Exact Match
assertNotNull	18,033 (16.1%)	79,476 (40.8%)	9203 (11.6%)
assertEquals	81,766 (72.8%)	79,680 (40.9%)	7217 (9%)
assertNull	10,484 (9.3%)	6,542 (3.4%)	1660 (25.3%)
assertSame	636 (0.6%)	5,791 (3%)	406 (7%)
assertFalse	651 (0.6%)	12,575 (6.5%)	43 (0.3%)
assertNotSame	334 (0.3%)	3,210 (1.6%)	89 (2.8%)
assertTrue	349 (0.3%)	3,268 (1.7%)	12 (0.37%)
syntx. incorrect	-	4,329 (2.2%)	-
Total:	112,253	194,871	18,630

18k are exact matches, indicating that the generated assertion are diverse relative to provided assertions in the dataset. We observe this diversity can come multiple ways. For example, the generated assertion may come from a different category, like the first and third examples in Figure 4, or may vary the assertion arguments, like the second example in Figure 4. We observe that ground truth oracles do not check all variables within the prefix which gives TOGLL an opportunity to generate diverse assertion oracles. We see this in the first example where the TOGLL generated assertion is requiring that sort yield the same array, whereas the EvoSuite assertions just checks the length of the array. Similarly the second example shows a TOGLL generated assertion for the `q1` variable, whereas EvoSuite’s assertion targets the `q2` variable. For the third example, TOGLL generates a different type of oracle. Not only are these TOGLL-generated assertions diverse, but they potentially add unique bug detection power which we explore in the next RQs. We can conclude that fine-tuning with EvoSuite oracles does not constrain TOGLL to EvoSuite-like oracles, as it generates distinct yet correct assertions involving various assertion types, variables, and expressions, with only a 9.5% exact match rate. In RQ4, we demonstrate that these distinct oracles detect 1,023 unique bugs that EvoSuite could not detect (Table III). Pre-trained on a vast codebase, TOGLL’s LLM backbone has likely encountered both developer-written and automated assertions. By leveraging this pretraining knowledge, TOGLL can generate diverse yet accurate assertions with strong bug detection effectiveness.

RQ3 Finding: TOGLL assertions are diverse with respect to both the assertion statement used and the variables and expressions targeted for observation in those assertions.

D. RQ4: Assessing Mutant Detection Effectiveness

Correct test oracles are essential for aligning with program requirements; however, correctness alone may not suffice to detect a diverse range of bugs. The strength of test oracles is

```

public void test3() throws Throwable {
    Object[] oA0 = new Object[0];
    Object[] oA1 = ArraySorter.sort(oA0, (Comparator<? super Object>) null);
    assertEquals(oA1, oA0); //TOGLL
    assertEquals(0, oA1.length); //Ground Truth
}

public void test9() throws Throwable {
    Quaternion q0 = Quaternion.I;
    Quaternion q1 = Quaternion.of(-0.0, (-118.4), -0.0, (-118.4));
    Quaternion q2 = Quaternion.subtract(q1, q0);
    assertEquals((-118.4), q1.getX(), 0.01D); //TOGLL
    assertEquals((-119.4), q2.getX(), 0.01); //Ground Truth
}

public void test14() throws Throwable {
    DateTime dT0 = DateTime.now();
    DateTime.Property dTP0 = dT0.yearOfEra();
    DateTimeField dTF0 = dTP0.getField();
    assertNotNull(dTF0); //TOGLL
    assertEquals("yearOfEra", dTF0.getName()); //Ground Truth
}

```

Fig. 4. Diverse yet correct test oracles generated by TOGLL

TABLE III
MUTANT DETECTION PERFORMANCE OF TOGLL, EvoSUITE AND TOGA
GENERATED ASSERTIONS.

Artifact	Bug Detected by			
	EvoSuite Assertion	EvoSuite Unique	TOGLL Assertion	TOGLL Unique
http	64	38	29	3
json	328	110	249	31
beanutils	551	46	516	11
collections4	248	104	162	18
dbutils	108	14	98	4
jsoup	598	177	509	88
imaging	1,869	629	1,315	75
lang3	2,301	397	2,063	159
configuration	271	51	266	46
jexl3	697	73	638	14
joda-time	1,545	446	1,323	224
net	747	175	592	20
pool2	155	8	150	3
spark	407	63	355	11
validator	602	52	560	10
scribejava	172	31	143	2
bcel	1,122	467	698	43
numbers	871	168	726	23
springside4	1,160	243	938	21
vfs2	339	37	310	8
rng	554	158	527	131
jcs3	621	81	548	8
async-http	69	10	59	0
weaver	20	2	18	0
Total:	15,985	3,690	13,318	1,023
		TOGA:	6,893	105

equally crucial. In RQ4, we investigate the mutant detection effectiveness of TOGLL-generated assertions and compare them with the SOTA EvoSuite and TOGA.

1) *Experimental Setup:* To set up this experiment, we generate three distinct test suites (T_{TOGLL} , T_{ES} , T_{IO}) for each of the 25 projects from the OracleEval25 dataset. T_{TOGLL} comprises test cases with TOGLL-generated assertions, excluding all test cases with non-compiling and incorrect (i.e., false positive) assertions produced by TOGLL. We recorded the set of final test cases, including the test class and the unique test identifier. For constructing T_{ES} , we use the same set of tests, but the test assertions were generated by EvoSuite. For T_{IO} , the same set of test cases was retained; however, this time, the test cases did not contain any test assertions. The

three test suites share identical counts of test cases, test case prefixes, and assertion totals. The unique aspect distinguishing them is the method used for generating assertions.

We utilize mutation testing to introduce a wide array of bugs into the source code of 25 projects. Unlike bug benchmarks such as Defects4J [36] or QuixBugs [45], which are limited in both the number and diversity of bugs, mutation test injects thousands of diverse bugs. We employ PIT [38], a state-of-the-art mutation testing tool for Java programs. PIT offers several advantages over other tools, as it generates meaningful mutants and fewer equivalent mutants [46]–[48]. We have used PIT-1.9.8 to generate a total of 69,793 strong mutants. For each mutated project, we run all three test suites to detect mutants. Mutants detected by T_{IO} are those that can be detected through implicit assertions, meaning test prefixes alone suffice for their detection without the need for any explicit assertions. Computing T_{IO} allows our study to follow the recommendations of [5] by removing mutants killed by implicit oracles from study results. We calculate the total number of mutants detected by EvoSuite, TOGLL and TOGA assertions as well as the unique mutants killed by assertions generated by each method.

2) *Results:* In Table III, Column 1 shows the names of 25 projects. Columns 2 and 3 present the total and unique mutants detected by EvoSuite-generated assertions, respectively. Columns 4 and 5 present the total and unique mutants detected by TOGLL assertions, respectively. Row 27 shows the total mutants detected across all projects by each method, whereas row 28 presents the performance of TOGA, in detecting total and unique mutants.

Out of the 69,793 mutants generated and covered by test cases, EvoSuite detected a total of 15,985 mutants, including 3,690 unique mutants not detected by TOGLL assertions. Conversely, TOGLL assertions identified 13,318 mutants, with 1,023 being unique mutants.

For all 25 projects, test cases were generated by EvoSuite, as discussed in Section II-B1. EvoSuite, a search-based dynamic approach, generates assertions based on executed program behavior and actual result values. Thus, it is expected to produce strong assertions capable of detecting a large number of mutants. TOGLL, in contrast, is a static approach that utilizes the test prefixes generated by EvoSuite, but lacks knowledge of any values computed during test execution. Despite this disadvantage, TOGLL-generated assertions detected 83% of the EvoSuite-detected mutants. This significantly outperformed TOGA, which detected only 43% of the EvoSuite-detected mutants under the same conditions.

TOGLL generated assertions were able to detect 1,023 mutants that EvoSuite missed, using the same set of test prefixes. This is also a significant improvement over TOGA, which only identified 105 unique mutants. In total, TOGLL killed nearly twice the number of mutants as TOGA, but when viewed relative to TOGA as a baseline TOGLL killed 9.7 times the number of unique mutants. This demonstrates that TOGLL can generate a significant number of strong test assertions and highlights TOGLL’s capability to complement

```
public int calculatePrintedLength(ReadablePeriod p, Locale l) {
    long vLong = getField(p);
    ..
    if (iFieldType >= SECONDS_MILLIS) {
        ..
        //Replaced long division with multiplication
        value = vLong / DateTimeConstants.MILLIS_PER_SECOND;
    }
    ..
    if (iSuffix != null) {
        sum += iSuffix.calculatePrintedLength((int)value);
    }
    return sum;
}

public void test327() throws Throwable {
    PFBBuilder.Literal pFL0 = PFBBuilder.Literal.EMPTY;
    Locale l0 = Locale.CHINA;
    Months m0 = Months.ZERO;
    int int0 = pFL0.calculatePrintedLength(m0, l0);
    assertEquals(0, int0); //TOGLL
    assertEquals("", l0.getVariant()); //EvoSuite
}

public Rad toRad() {
    return this; //Null Return
}

public void test13() throws Throwable {
    Angle.Rad angle_Rad0 = Angle.Rad.PI;
    Angle.Rad angle_Rad1 = angle_Rad0.toRad();
    assertNotNull(angle_Rad1); //TOGLL
    assertEquals(1.57, Angle.PI_OVER_TWO, 0.01); //EvoSuite
}
```

Fig. 5. TOGLL generated assertions detecting unique mutants that EvoSuite generated assertions mean not.

EvoSuite’s assertions.

We perform a qualitative study of the TOGLL assertions that could kill unique mutants. This study was opportunistic in that we look into the 1023 unique mutants, their corresponding mutated MUTs from smallest to largest and stopped when we found a handful of examples that illustrated oracle diversity. Figure 5 shows two examples from the joda-time and Apache commons-angle package. In the first example, `test327` calls the `calculatePrintedLength` method which was mutated by ‘replacing long division with multiplication’, TOGLL checks the return variable `int0` and is able to kill the mutant. EvoSuite checks a value that is independent of the execution of the MUT and has no chance to kill the mutant. The second example, TOGLL check that `angle_Rad1` should not be null and thus was able to detect the mutant which replaces the return value with null. Here again the EvoSuite generated assertion checks a value that is independent of the MUT. While anecdotal these examples illustrate how TOGLL leverages the EvoSuite generated test prefixes and produces valuable test oracles that EvoSuite can not.

RQ4 Finding: TOGLL generated test oracles kill 17% fewer mutants than EvoSuite tests, despite having less information about test behavior. Relative to TOGA, TOGLL yields nearly a 10-fold increase in mutant kills - thereby establishing a new SOTA in neural oracle generation.

E. RQ5: Assessing Real Bug Detection Effectiveness

In RQ4, we evaluate TOGLL’s bug detection effectiveness using mutants. In this research question, we investigate

TABLE IV

COMPARISON OF ORACLE CLASSIFICATION PERFORMANCE BETWEEN TOGLL AND TOGA ON THE DEFECTS4J DATASET. METRICS INCLUDE PRECISION (P), RECALL (R), F1-SCORE (F1), AND ACCURACY (ACCU.).

Approach	Exception			Assertion			Overall		
	P	R	F1	P	R	F1	Accu.	P	R
TOGA	.14	.12	.13	.83	.86	.84	.74	.49	.49
TOGLL	1	1	1	1	1	1	1	1	1

TOGLL’s real bug detection effectiveness and compare its performance with TOGA

1) *Experimental Setup*: To conduct this study, we utilize Defects4J [36], a benchmark dataset consisting of real Java bugs. To ensure a fair comparison with TOGA, we follow the exact same experimental setup: the same Docker container environment, the same input dataset, and the exact same set of scripts provided in the TOGA replication package [49]. The Defects4J dataset consists of 374 input samples, each with a MUT, a test prefix, and Javadoc documentation. Of these, 304 require an assertion and 70 require an exception oracle. For this dataset, we generate oracle predictions using our method, TOGLL. Next, TOGLL generated test oracles are integrated with the respective test prefixes. These integrated test cases (test prefix + oracle) are then executed on both the buggy and fixed program versions. A bug is considered detected if a test passes on the fixed version but fails on the buggy version, indicating that the test oracle correctly aligns with the expected behavior and is capable of detecting the buggy behavior. To understand the results better, we first present the oracle classification performance of both methods in Table IV, followed by their bug detection performance in Table V.

2) *Results*: Table IV presents the oracle classification performance of both methods on the Defects4J dataset. In this classification task, each method determines which type of oracle is needed for a given input prefix. The table shows performance metrics for both classes: exception and assertion, as well as the overall metrics.

TOGA exhibits very poor precision, recall, and F1-score for the class of test prefixes requiring an exception oracle. In contrast, our method, TOGLL, demonstrates a very high classification accuracy. In RQ2, the exception oracle classification accuracy averaged 93.4% for 222k input samples. In this smaller study of 374 inputs it achieved an accuracy of 100%.

Regarding assertion oracles, although TOGA correctly classified that an assertion oracle is needed for 84% of the inputs, it generated an explicit assertion for only 40% of them. For 60% of inputs, TOGA could not generate any assertions. On the other hand, TOGLL correctly classified that an assertion oracle is needed with 100% accuracy for the Defects4J study, and for 97% of them, it generated an assertion oracle.

Table V shows the bug detection performance of both approaches. Column 2 shows the total number of bugs detected by exception oracles. TOGLL detected a total of 27 bugs with its exception oracles, whereas TOGA detected only 5. These

TABLE V

DEFECTS4J BUG DETECTION PERFORMANCE OF TOGLL VS. TOGA. A BUG CAN BE DETECTED BY DIFFERENT TYPE OF ORACLES.

Approach	Bug Detected By				FP
	Exception Oracle	Explicit Assertion Oracle	EvoSuite Test Prefix Only	Total Unique	
TOGA	5	26	36	56	0.45
TOGLL	27	37	1	65	0.42

findings align with the data in Table IV, which shows that TOGA’s exception oracle generation precision is only 0.14. These results are also consistent with the RQ2 findings, which show a significant performance difference between TOGA and TOGLL, indicating a similar trend across multiple datasets.

Column 3 shows the total number of bugs detected by explicit assertion oracles (non-empty assertions). For the input test prefixes requiring an assertion oracle, TOGA generated assertions for only 40% of the inputs (121 out of 304), detecting a total of 26 bugs. In contrast, TOGLL generated assertions for 97% of the inputs, detecting a total of 37 bugs.

Column 4 shows the number of bugs detected by test prefixes alone. TOGA could not generate assertions for 183 out of 304 prefixes. Simply running the test prefixes on the buggy versions detected 36 bugs – these are detected by the implicit oracles of the Java Runtime System that expects no uncaught exception. TOGLL did not generate an assertion for only 9 prefixes, with only 1 bug detected by running those prefixes.

As suggested by recent prior work [5], [50], bugs detected by implicit oracles when executing test prefixes throwing uncaught exceptions should *not* be considered as contributions of a test oracle generation method. We followed this guidance in RQ4 and applying it here by excluding bugs found in this way reveals that TOGLL detects 64 bugs, which is 106.5% more bugs than the 31 bugs detected by TOGA. Even if one were to count the bugs detected by implicit oracles, TOGLL detects 16% more bugs than TOGA on Defects4J.

RQ5 Finding: When excluding the bugs that are detected by test prefixes alone, TOGA detected a total of 31 Defects4J bugs with its exception and explicit assertion oracles. In comparison, TOGLL detected 64 bugs and improvement of 106%.

F. Threats to Validity

In our large-scale study, we have utilized open-source and widely recognized SF110 dataset [25]. Yet, our results may not generalize across other datasets. To ensure generalizability, we further evaluated TOGLL on the unseen OracleEval25 [5] and Defects4J [36] dataset.

In our comprehensive study, we created several tools and scripts which may contain bugs. We used publicly available

libraries to mitigate the risk and conducted extensive validity tests and repeated our experiments to ensure consistent results.

In our investigation, during fine-tuning we utilized the ‘exact match’ metric to compute oracle generation accuracy. This is a widely used metric by the research community. For unseen inference, we relied on test validation. This methodology, we assert, offers a reliable accuracy metric. Additionally, to complement the results from mutation testing, we performed a real bug detection study. Both metrics are well-established and used in many research studies [4], [5], [7].

IV. LIMITATION AND FUTURE DIRECTION

In this work, we have demonstrated the potential of LLMs in generating correct, diverse and strong test oracles that can effectively detect large number of unique bugs. However, there are challenges which require further research efforts.

A. Compilation Error

Our in-depth experimental results show that TOGLL generates assertions that are non-compiling about 5% of the time. Among them we have seen assertions that are ‘garbage’ strings and one’s that have minor syntax or type errors. In future work, we want to fine-tune LLMs to generate syntax and type correct assertion using grammar and type based constraints similar to [17]. Additionally, incorporating compilation errors into a refinement prompt to improve the the generated oracle is another strategy worth pursuing [51].

B. Token Length Limits

We have seen that LLMs can not process input samples that exceeds the maximum token length capacity of the LLMs. In our study, a total of 3% of the samples exceeded the threshold of 600 tokens when using P6 that includes MUT, docstring and prefix. Rather than increasing the token threshold, our study suggests that in such cases we could shift to a more concise prompt, such as P3 which replaces the MUT with the method signature without significantly compromising performance.

C. False Positives

One important and long standing issue with all type of learning based oracle generation method is that they can generate false positives, i.e., oracles that fail on a correct program. TOGLL exhibited a 7% false positive rate for exception oracles and a 25% rate for assertion oracles which is a significant improvement over the respective rates for TOGA – 81% and 47%. We have studied a sample of false positive assertions and observed that many of them include numeric literals which is the source of the false positive. Based on these observations, we conjecture that assertions relating program variables, or expressions involving variables, may be less susceptible to false positives. In future research, we explore the benefits of using high quality documentation, which may capture such relations, and develop prompting strategies that encourage the generation of assertions that are not as dependent on numeric literals.

V. RELATED WORK

EvoSuite [24] is a state of the practice automated test oracle generation that uses a search-based method. EvoSuite generates oracles by observing test execution and capturing values that it includes in assertions. This creates regression oracles that are valuable for detecting undesirable changes in program behavior. Importantly for our study, by construction these oracles capture the behavior of the system under test making them a useful baseline for judging correct test oracles.

Randoop [52] is feedback-directed random testing for Java. Research has explored the generation of both assertion [9] and exception oracles [10] for Randoop tests using NLP approaches. Subsequent NLP-based methods, such as ATLAS [14] and AthenaTest [16], have been shown to outperform these methods and more recently TOGA [17] established a new state of the art by a large margin. This is why we use TOGA as baseline in our study.

In the past year, techniques for test generation using LLMs have emerged. For example, TESTPILOT [21], an LLM-based test generation tool for JavaScript that automatically generates unit tests for JavaScript. TESTPILOT achieved 70% statement coverage and 52.8% branch coverage, which outperformed the state-of-the feedback-directed JavaScript test generation technique, Nessie [53]. A recent study [54] that utilized LLMs for generating test for the SF110 large-scale real world Java projects showed that LLMA could only achieve 2% coverage, whereas EvoSuite can achieve more than 90% coverage. This suggests that LLMs are not very effective in generating test for real world Java program, which is why in our study instead of focusing on both test input and test oracle generation, we focus on a single task test oracle generation with LLM.

VI. CONCLUSION

We conduct the first large-scale investigation of the ability of large language models to automatically generating correct and strong test oracles. To this end, we fine-tune seven code LLMs with six different prompts with varying level of information. Our findings reveal that effective prompt design can improve the accuracy of the models significantly and that with effective prompts smaller size model can also perform equally or even better than larger models.

With the best performing model and prompt we develop TOGLL, our LLM-based automated test oracle generation method. We extensively evaluate TOGLL through large-scale study to investigate its generalizability and capability to generate correct, diverse and strong test oracles. Our experimental study answers critical research questions, demonstrating that TOGLL can generate 3.8x more correct assertion and 4.9x more exception oracles than the previous SOTA neural method, TOGA. Generating correct oracle does not necessarily mean they are diverse and strong enough to detect bugs [55]. We find that TOGLL is capable of generating diverse test oracle where only 9.5% are an exact match with the ground truth. Furthermore, TOGLL’s diverse assertions can detect a large number of unique bugs that EvoSuite can not detect. For

both mutant and real bug detection, TOGLL significantly outperformed TOGA, establishing itself as the new SOTA.

We describe remaining challenges and discuss several actionable future directions. We believe our large-scale study and in-depth investigation lay down the groundwork for further advancements in automated test oracle generation.

VII. ACKNOWLEDGMENT

This material is based in part upon work supported by National Science Foundation awards 2129824 and 221707. The authors acknowledge Research Computing at The University of Virginia for providing computational resources and technical support that have contributed to the results reported within this publication.

REFERENCES

- [1] Synopsys Editorial Team, “Coverity report on the ‘goto fail’ bug,” blog post, Synopsys, Mountain View, CA, Feb. 25, 2014; <http://security.coverity.com/blog/2014/Feb/a-quick-post-on-apple-security-55471-aka-goto-fail.html>.
- [2] H. News, “Twitter outage report,” 2016, <https://news.ycombinator.com/item?id=8810157>.
- [3] A. M. Porrello, “Death and denial: The failure of the therac-25, a medical linear accelerator,” *Death and Denial: The Failure of the THERAC-25, A Medical Linear Accelerator*, 2012.
- [4] S. B. Hossain, M. B. Dwyer, S. Elbaum, and A. Nguyen-Tuong, “Measuring and mitigating gaps in structural testing,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1712–1723.
- [5] 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*, ser. ESEC/FSE 2023. NY, USA: Association for Computing Machinery, 2023, p. 120–132. [Online]. Available: <https://doi.org/10.1145/3611643.3616265>
- [6] D. Schuler and A. Zeller, “Checked coverage: an indicator for oracle quality,” *Software testing, verification and reliability*, vol. 23, no. 7, pp. 531–551, 2013.
- [7] Y. Zhang and A. Mesbah, “Assertions are strongly correlated with test suite effectiveness,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 214–224. [Online]. Available: <https://doi.org/10.1145/2786805.2786858>
- [8] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.
- [9] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, “Translating code comments to procedure specifications,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 242–253. [Online]. Available: <https://doi.org/10.1145/3213846.3213872>
- [10] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, “Automatic generation of oracles for exceptional behaviors,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 213–224. [Online]. Available: <https://doi.org/10.1145/2931037.2931061>
- [11] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, “Memo: Automatically identifying metamorphic relations in javadoc comments for test automation,” *Journal of Systems and Software*, vol. 181, p. 111041, 2021. [Online]. Available: <https://doi.org/10.1016/j.jss.2021.111041>
- [12] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, “Inferring method specifications from natural language api descriptions,” in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 815–825.
- [13] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, “@tcomment: Testing javadoc comments to detect comment-code inconsistencies,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.
- [14] 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. [Online]. Available: <https://doi.org/10.1145/3377811.3380429>
- [15] 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. [Online]. Available: <https://doi.org/10.1145/3524481.3527220>
- [16] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, “Unit test case generation with transformers and focal context,” *arXiv preprint arXiv:2009.05617*, 2020.
- [17] 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*, ser. ICSE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2130–2141. [Online]. Available: <https://doi.org/10.1145/3510003.3510141>
- [18] Z. Li, C. Wang, Z. Liu, H. Wang, D. Chen, S. Wang, and C. Gao, “Cctest: Testing and repairing code completion systems,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1238–1250.
- [19] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, “Using an llm to help with code understanding,” in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 881–881.
- [20] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, “Inferfix: End-to-end program repair with llms,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.
- [21] 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*, vol. 50, no. 1, pp. 85–105, 2024.
- [22] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “Codamos: Escaping coverage plateaus in test generation with pre-trained large language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.
- [23] M. L. Siddiq, J. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, “Exploring the effectiveness of large language models in generating unit tests,” *arXiv preprint arXiv:2305.00418*, 2023.
- [24] 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. [Online]. Available: <https://doi.org/10.1145/2025113.2025179>
- [25] —, “A large-scale evaluation of automated unit test generation using evosuite,” *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, dec 2014. [Online]. Available: <https://doi.org/10.1145/2685612>
- [26] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *arXiv preprint arXiv:2308.10620*, 2023.
- [27] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, “On the dangers of stochastic parrots: Can language models be too big?” in *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, 2021, pp. 610–623.
- [28] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, “A deep dive into large language models for automated bug localization and repair,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: <https://doi.org/10.1145/3660773>
- [29] Hugging Face, “Hugging face: The ai community building the future,” <https://huggingface.co/>, 2024, accessed: 2024-05-1.
- [30] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *arXiv preprint arXiv:2102.04664*, 2021.
- [31] CodeParrot, “<https://huggingface.co/codeparrot/codeparrot-small-multi>.”
- [32] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” 2023.
- [33] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” 2022.

- [34] Phi-1, “<https://huggingface.co/microsoft/phi-1>.”
- [35] A. C. Proper, “Apache commons proper – a repository of reusable java components,” 2022, <https://commons.apache.org/components.html>, Last accessed on 2022-10-11.
- [36] 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. [Online]. Available: <https://doi.org/10.1145/2610384.2628055>
- [37] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.
- [38] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “Pit: a practical mutation testing tool for java,” in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452. [Online]. Available: <https://doi.org/10.1145/2931037.2948707>
- [39] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, “Practical mutation testing at scale: A view from google,” *IEEE Transactions on Software Engineering*, 2021.
- [40] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 654–665. [Online]. Available: <https://doi.org/10.1145/2635868.2635929>
- [41] J. Andrews, L. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments? [software testing],” in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005, pp. 402–411.
- [42] G. Petrović, M. Ivanković, G. Fraser, and R. Just, “Does mutation testing improve testing practices?” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 910–921. [Online]. Available: <https://doi.org/10.1109/ICSE43902.2021.00087>
- [43] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.
- [44] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” 2024.
- [45] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 55–56. [Online]. Available: <https://doi.org/10.1145/3135932.3135941>
- [46] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, “How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults,” *Empirical Software Engineering*, vol. 23, no. 4, pp. 2426–2463, 2018.
- [47] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque, “Assessing and improving the mutation testing practice of pit,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 430–435.
- [48] S. Rani, B. Suri, and S. K. Khatri, “Experimental comparison of automated mutation testing tools for java,” in *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*, 2015, pp. 1–6.
- [49] E. Dinella, G. Ryan, S. K. Lahiri, and T. Mytkowicz, “Replication Artifact for TOGA: A Neural Method for Test Oracle Generation,” Feb. 2022.
- [50] Z. Liu, K. Liu, X. Xia, and X. Yang, “Towards more realistic evaluation for neural test oracle generation,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 589–600.
- [51] P. Deligiannis, A. Lal, N. Mehrotra, and A. Rastogi, “Fixing rust compilation errors using llms,” *arXiv preprint arXiv:2308.05177*, 2023.
- [52] 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.
- [53] E. Arteca, S. Harner, M. Pradel, and F. Tip, “Nessie: Automatically testing javascript apis with asynchronous callbacks,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1494–1505.
- [54] M. L. Siddiq, J. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, “Exploring the effectiveness of large language models in generating unit tests,” *arXiv preprint arXiv:2305.00418*, 2023.
- [55] S. B. Hossain, “Ensuring critical properties of test oracles for effective bug detection,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 176–180. [Online]. Available: <https://doi.org/10.1145/3639478.3639791>