

Doc2OracLL: Investigating the Impact of Documentation on LLM-based Test Oracle Generation

SONEYA BINTA HOSSAIN, University of Virginia, USA

RAYGAN TAYLOR, Dillard University, USA

MATTHEW DWYER, University of Virginia, USA

Code documentation is a critical artifact of software development, bridging human understanding and machine-readable code. Beyond aiding developers in code comprehension and maintenance, documentation also plays a critical role in *automating various software engineering tasks, such as test oracle generation (TOG)*. In Java, Javadoc comments offer structured, natural language documentation embedded directly within the source code, typically describing functionality, usage, parameters, return values, and exceptional behavior. While prior research has explored the use of Javadoc comments in TOG alongside other information, such as the method under test, their potential as a stand-alone input source, the most relevant Javadoc components, and guidelines for writing effective Javadoc comments for automating TOG remain less explored.

In this study, we investigate the impact of Javadoc comments on TOG through a comprehensive analysis. We begin by fine-tuning 10 large language models using three different prompt pairs to assess the role of Javadoc comments alongside other contextual information. Next, we systematically analyze the impact of different Javadoc comment's **components** on TOG. To evaluate the generalizability of Javadoc comments from various sources, we also generate them using the GPT-3.5 model. We perform a thorough bug detection study using *Defects4J* dataset to understand their role in real-world bug detection. Our results show that incorporating Javadoc comments improves the accuracy of test oracles in most cases, aligning closely with ground truth. We find that Javadoc comments *alone* can achieve comparable or even better performance when using the implementation of MUT. Additionally, we identify that the **description** and the **return tag** are the most valuable components for TOG. Finally, our approach, when using only Javadoc comments, detects between 19% and 94% more real-world bugs in Defects4J than prior methods, establishing a new state-of-the-art. To further guide developers in writing effective documentation, we conduct a detailed qualitative study on when Javadoc comments are helpful or harmful for TOG.

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

Additional Key Words and Phrases: software testing, test oracles, documentation, large language model

ACM Reference Format:

Soneya Binta Hossain, Raygan Taylor, and Matthew Dwyer. 2025. Doc2OracLL: Investigating the Impact of Documentation on LLM-based Test Oracle Generation. 1, 1 (March 2025), 23 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 INTRODUCTION

Testing is a critical step in the software development process. It involves solving two challenges: (1) *selecting test inputs* to adequately exercise program behavior, and (2) judging whether the program produces the desired result for those inputs – the *test oracle* problem [3]. The past decade has

Authors' addresses: Soneya Binta Hossain, University of Virginia, Charlottesville, USA, sh7hv@virginia.edu; Raygan Taylor, Dillard University, New Orleans, USA, raygan.taylor@dillard.edu; Matthew Dwyer, University of Virginia, Charlottesville, USA, md3cn@virginia.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/authors. Publication rights licensed to ACM.

ACM XXXX-XXXX/2025/3-ART

<https://doi.org/XXXXXXX.XXXXXXX>

| Javadoc Comment | MUT | Test Oracles |
|--|---|---|
| <pre> /** * Gets the time zone for the given ID. * * @param id the time zone ID, not null * @return the DateTimeZone object for the ID * @throws IllegalArgumentException if ID * not recognized */ public static DateTimeZone forID(String id); </pre> | <pre> public static DateTimeZone forID(String id) { if (id == null) { return null; } // check if id is a valid time zone ID if (id.equals("UTC")) { return UTC; } return null; //should throw an exception } </pre> | <pre> public void testForID() { DateTimeZone zone = DateTimeZone.forID("Invalid/Zone"); assertNull(zone); // Incorrect oracle } public void testForIDUsingJavadoc() { try { DateTimeZone.forID("Invalid/Zone"); fail("Expected IllegalArgumentException"); } catch (IllegalArgumentException e) { // Correct behavior } // Correct oracle } </pre> |

Fig. 1. Incorrect oracle generated from buggy MUT and correct oracle generated from Javadoc comments.

witnessed enormous advances in automating support for the first of these [7, 13, 25, 31] and such techniques are now regularly used in practice.

Providing effective automated support for the oracle problem has been less successful, though in recent years, researchers have made some progress using machine learning (ML) techniques [10, 11, 17, 37]. All of these techniques either require [10, 37] or allow [11, 18] for the implementation of the method under test (MUT) to be included in the ML model’s input to generate a test oracle. Recent research [26] has pointed out that this may negatively impact the quality of the test oracle. For example, if the implementation contains a bug, the model may encode that buggy behavior into the oracle, resulting in an inappropriately passing test case. More generally, this approach tends to generate oracles that are primarily useful in a regression context.

One way to address this weakness is to leverage good documentation for the MUT. Conceptually, good method-level comments provide a correct and complete, yet abstract, description of intended behavior that is decoupled from implementation detail [2]. Such code comments are widely acknowledged by practitioners as useful for “development and testing” tasks [1]. Moreover practitioners find that method-level comments are the most valuable of all forms of software documentation and, for such comments, information about “functionality”, “usage”, and “input & output” is most important [21]. We observe that this type of information aligns with the needs of software testers, who aim to select test inputs based on appropriate method usage and define test oracles to assess whether outputs match the intended functionality.

Figure 1 shows an example from the Joda-Time library that illustrates the risk of using the MUT and the benefit of using Javadoc instead. The MUT returns null when the time zone id is not recognized, where as based on the Javadoc comments, it should throw an IllegalArgumentException. When using the MUT code to generate test oracles, it generates oracles based on the faulty MUT implementation and is thus unable to detect the bug. When using the Javadoc comments only, the generated oracle expects the correct behavior that an IllegalArgumentException should be thrown.

In this paper, we study the value of method-level documentation in test oracle generation (TOG) using large-language models (LLM). We use Javadoc comments as method-level documentation, which are specifically designed to document Java classes, methods, and interfaces. These comments are written in a structured format using natural language, documenting detailed information about the expected behavior of method, its parameters and return value, and exceptional behavior. Their structured format, combined with rich contextual information, makes Javadoc comments highly suitable for LLMs. The first sentence of each Javadoc comment consist of a concise but complete description of the behavior of the MUT – we refer to this as the description. After that, Javadoc comment may include @param tag with parameter description, @return tag with return value

details, `@throws/@exception` tag detailing exceptional behavior, and some other tags, such as, the `@see` tag which introduces cross-references to other methods [30].

We build on recent work [17] that explored a range of prompt formats to fine-tune LLMs for test oracles generation task. Unlike that work, our focus here is on the role of the Javadoc in test oracle generation. While [17] did evaluate the differential benefit of including Javadoc comments in prompts, that work did not account for the fact that 60% of the dataset used in their evaluation had no Javadoc. This caused their study to substantially underestimate the benefit of leveraging Javadoc in test oracle generation. Furthermore, their work did not explore the impact of various components of Javadoc comments on TOG, nor did they examine the role of Javadoc comments in bug detection.

We address this by developing several dataset variants in which every MUT has Javadoc comments. The first dataset variant selects more than 55-thousand samples for which developer written Javadoc comments are available. It has been well-documented that the quality of method-level documentation varies significantly in practice [42], but short of human subject studies, e.g., [1, 2, 21], determining documentation quality is very challenging especially in light of recent results demonstrating the unreliability of natural language similarity metrics for this problem [35]. Consequently, we generate a second dataset variant, inspired by research on code summarization [24, 28], that generates Javadoc comment that summarizes the MUT using a generative model. Studies using these two dataset variants show that incorporating Javadoc comments for the test oracle generation task leads to a 10-20% improvement in the ability to produce ground-truth test oracles, both assertion and exception.

Javadoc is a rich documentation format with many different tag types as shown in Figure 1 and LLMs produce higher-quality responses when prompted with the most relevant information for a task. This led us to conduct an ablation study on the structure of Javadoc comments in which we systematically dropped the description and tags to understand their differential value in the TOG task. We find clear evidence that the description and `@return` tag provide the greatest value, which suggests that when LLM token limits come into play in prompting prioritizing these Javadoc elements in a prompt is critical for high-quality TOG.

Finally, while the bulk of our study is conducted on a large and broadly representative dataset, we go further to understand how fine-tuned LLMs perform TOG generation on unseen datasets that harbor real-world bugs. In a study on Defects4J [23], we demonstrate that our methods generate test oracles that are able to detect up to 44% more bugs than prior approaches [10, 11], while not leveraging the MUT implementation. When prior approaches do not use the MUT their performance degrades and our method can detect 94% more bugs.

The primary contributions of this paper are: (1) **Empirical demonstration** across a wide range of fine-tuned LLMs and two distinct documentation sources, showing that method-level documentation improves TOG’s ability to match ground truth oracles. (2) **Ablation study** on Javadoc comment structure, identifying **descriptions** and **@return** tags as the most valuable components for TOG. (3) **Bug detection analysis**, showing that fine-tuned models leveraging Javadoc comments can generate test oracles that detect significantly more bugs than prior TOG approaches relying on MUT code. (4) **Qualitative analysis** providing guidelines on writing effective documentation to enable the generation of stronger test oracles.

Collectively, these contributions provide compelling evidence that TOG can be performed using software artifacts that exclude the method implementation, reducing a major source of bias in prior work.

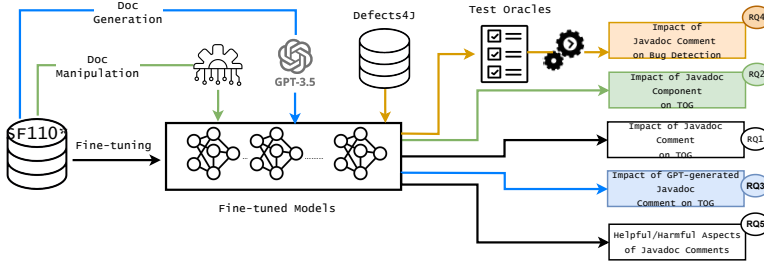


Fig. 2. Overview of our approach Doc2OracLL.

2 APPROACH

In Figure 2, we show an overview of our approach. We first fine-tune several large language models, as discussed in Section 2.1.3, using three pairs of prompts described in Section 2.1.2. To fine-tune the LLMs on these prompts, we employ the dataset detailed in Section 2.1.1. We generate various version of the test dataset by manipulating developer written Javadoc comments and by generating alternative Javadoc using GPT-3.5 model. We also perform unseen inference in the context of a real-bug detection study to investigate the impact of Javadoc comments. In this section, we provide an overview of the dataset, the prompts, the LLMs, and the metrics used to evaluate our method and compare it with other approaches.

2.1 Supervised Fine-tuning

This section discusses the dataset, prompts, and LLMs that are fine-tuned to investigate the impact of Javadoc comments on the test oracle generation (TOG) task.

2.1.1 SF110* Dataset. To construct our dataset, we utilized the SF110 benchmark [13], which comprises 110 real-world open-source Java projects from various domains, all sourced from the SourceForge repository. The benchmark contains a total of 23,000 Java classes and is widely used for unit test generation [17, 38]. The dataset consists of tuples $((p_i, m_i, d_i), o_i)$, where p_i is the test prefix, m_i is the MUT, d_i is the Javadoc comments (if available), and o_i is the ground truth test oracle, either an assertion or an exception. In total, the dataset includes 140,514 samples, of which 55,575 (40%) contain Javadoc comments. These samples with Javadoc comments are used for fine-tuning the LLMs and have been split into three distinct subsets: 90% for training, 5% for validation, and 5% for testing.

2.1.2 Prompts. The primary goal of our study is to investigate the impact of Javadoc comments on automated TOG. To ensure our experiments are *well-controlled* and to *isolate* the effect of Javadoc comments, we designed three pairs of prompts. In each prompt pair, the only *difference* between the two prompts is *whether Javadoc comments are present or absent*.

- **Pair 1 ($\mathcal{P}_1, \mathcal{P}_2$):** In this prompt pair, both prompts share the same minimal input—namely, the test prefix. However, prompt \mathcal{P}_2 also includes the Javadoc comments. By comparing the accuracy of these two prompts, we can directly assess the impact of Javadoc comments when only minimal context—the test prefix—is provided.
- **Pair 2 ($\mathcal{P}_3, \mathcal{P}_4$):** In this prompt pair, both prompts include two key pieces of information—the test prefix and the MUT signature—while prompt \mathcal{P}_4 adds the Javadoc comments as extra context. By comparing the accuracy of these two prompts, we can gauge the influence of Javadoc comments when both the test prefix and MUT signature are provided. Compared to

Pair 1, this pair further elucidates how additional contextual information (the MUT signature) affects the role and impact of Javadoc comments.

- **Pair 3 ($\mathcal{P}_5, \mathcal{P}_6$):** In this prompt pair, both prompts include the test prefix and the full MUT (Method Under Test) code. However, prompt \mathcal{P}_6 also provides additional Javadoc comments. By comparing the performance of these two prompts, we can analyze the contribution of Javadoc comments when the model already has access to the complete MUT code. This comparison clarifies the added value of Javadoc comments when substantial contextual information is already available from both the test prefix and the MUT code, further refining our insights from prompt pairs 1 and 2.

2.1.3 Code Models. Decoder-only models are widely used for code generation [20], and smaller models offer several advantages over larger ones. For instance, fine-tuning smaller models generally requires fewer GPU resources, making them more cost-effective and reducing their environmental impact [4]. In this study, we selected four smaller models (fewer than 1 billion parameters), four models ranging from 1 to 2.7 billion parameters, and two models with 7 billion parameters. All of these models are decoder-only, pre-trained on multiple programming languages, and publicly available on Hugging Face[22]. They also come with sufficient documentation for fine-tuning, which has led to their adoption across a variety of tasks throughout the software development lifecycle (SDLC)[19, 20].

CodeGPT [27]: In our study, we used the *microsoft/CodeGPT-small-java* model, a GPT-style language model with 110 million parameters. This model was trained and tested on a dataset of Java programs and is pre-trained for the code completion task, which aligns well with our designed prompts.

CodeParrot [9]: We fine-tuned the 110-million-parameter version of CodeParrot, which was originally pre-trained for code generation on nine different programming languages, including Java, C++, and Python. Specifically, we employed the *codeparrot/codeparrot-small-multi* model from Hugging Face. Given its strong generalization on unseen data and its ability to outperform larger models [17], we selected this model for the TOG task.

CodeGen [29]: CodeGen comprises four autoregressive language models of varying sizes (350M, 2B, 6B, and 16B). In this paper, we fine-tuned the *Salesforce/codegen-350M-multi* and *Salesforce/codegen-2B-multi* models, each pre-trained on a dataset spanning six programming languages: C, C++, Go, Java, JavaScript, and Python. We selected these models for the TOG task due to their strong code comprehension and generation capabilities.

PolyCoder [44]: PolyCoder is a family of large language models available in three parameter sizes—400M, 2.7B, and 16B—trained on 249 GB of code across 12 programming languages. In this work, we fine-tuned the *NinedayWang/PolyCoder-0.4B* and *NinedayWang/PolyCoder-2.7B* models for oracle generation.

Phi-1 [33]: Phi-1 is a 1.3-billion-parameter, decoder-only transformer model originally designed for Python code generation. The *microsoft/phi-1* variant was pre-trained on Python code from multiple sources. Although its primary focus is Python, our study found that it performs comparably to other models when fine-tuned for Java code generation. Nonetheless, despite achieving good results after fine-tuning, the model struggled to generalize effectively to unseen Java code.

CodeLlama [8]: Code Llama is a family of pre-trained and fine-tuned generative models, ranging in size from 7 billion to 34 billion parameters. In our study, we used the *codellama/CodeLlama-7b-hf* model—a 7-billion-parameter, decoder-only language model designed for general code synthesis and comprehension. We selected Code Llama for its strong performance on Java code infilling tasks, where it has outperformed larger models [36].

CodeGemma [15]: CodeGemma is a collection of lightweight models designed for tasks such as code completion, generation, natural language understanding, mathematical reasoning, and instruction following. The collection includes a 7-billion-parameter model pre-trained for code completion and generation, a 7-billion-parameter instruction-tuned model for code chat and instruction following, and a 2-billion-parameter variant optimized for quick code completion. In this study, we fine-tuned both the 2-billion and 7-billion pre-trained models (*google/codegemma-2b* and *google/codegemma-7b* from Hugging Face [22]) for test oracle generation.

2.1.4 Metric. To compute the accuracy of the generated test oracles, we use *exact match* with the ground truth oracle as our metric, a common approach in machine learning research [17, 19]. However, we acknowledge that this approach may *underestimate* accuracy, since a test oracle can be correct even if it does not match the ground truth exactly. For example, we observed assertion oracles like `assertFalse(boolean0)` and `assertEquals(false, boolean0)`, which are semantically equivalent despite differing in form.

2.2 Unseen Inference

To investigate the generalizability of Javadoc comments' impact on TOG, we perform an unseen inference experiment.

2.2.1 Dataset and Metric. For this unseen experiment, we use the widely recognized **Defects4J** dataset [23], which contains 835 bugs from 17 real-world Java projects. Each bug (1) is recorded in the project's issue tracker, (2) involves source code changes, and (3) is reproducible. The dataset provides both buggy and fixed versions of the code. The fixed version applies a minimal patch that resolves the bug, ensuring all project tests pass, while the buggy version fails at least one test. Additionally, Defects4J includes utilities for generating and evaluating test suites, allowing researchers to confirm whether generated tests pass on fixed versions and detect bugs in buggy versions. This dataset is widely used for evaluating automated test oracle generation and program repair methods [11, 17–19].

In our study, we *execute the test cases along with our method-generated test oracles* on both the buggy and the fixed versions of each program. To evaluate bug detection effectiveness, we measure (1) the total number of bugs detected by our method, (2) the unique number of bugs detected by each LLM and prompt pair, (3) both individually and in combination, (4) and, finally, we also compare these results with two baseline approaches: *TOGA* [10] and *nl2postcondition* [11].

3 EXPERIMENTAL STUDY

We examine several key factors: the effects of Javadoc components on TOG as stand-alone or complementary input, the contribution of various Javadoc components to TOG, the effectiveness of GPT-generated Javadoc comments, the role of Javadoc comments in detecting real-world bugs, and when Javadoc comments can positively/negatively impact TOG. We answered following *five* research questions:

RQ1: What is the impact of developer-written Javadoc comments on the performance of different LLMs in TOG? *In almost all cases, including Javadoc comments improves TOG performance, often by more than 20% when the base prompt contains minimal information.*

RQ2: What is the contribution of each Javadoc comment component to the accuracy of TOG? *We observe that the description and @return tags provide the most value when using Javadoc comments for TOG.*

RQ3: What is the impact of GPT-generated Javadoc comments on TOG performance? *Even GPT-generated Javadoc comments can improve performance by about 10% compared to using no Javadoc comments.*

RQ4: What is the impact of Javadoc comments in detecting real-world bugs from Defects4J? *When Javadoc comments are included, our method significantly outperforms existing SOTA approaches. Notably, using Javadoc comments alone can yield better results than using the MUT implementation.*

RQ5: What aspects of Javadoc comments make them effective or ineffective for TOG? *Javadoc comments are effective when they clearly specify expected behavior, input/output contracts, and return values, but ineffective when they are lengthy, vague/incomplete, or irrelevant.*

3.1 RQ1: Impact of Javadoc Comments on TOG

This study investigates the impact of Javadoc comments as a standalone or complementary input source for TOG. We evaluate 60 model-prompt combinations to systematically assess *how javadoc comments influence TOG performance across different LLMs and prompt contexts.*

3.1.1 Experimental Setup. We fine-tune 10 LLMs using the SF110* dataset (detailed in Section 2.1.1), where each input sample includes the method under test (MUT), a test prefix, and associated Javadoc comments. To isolate the impact of Javadoc comments, we design three pairs of prompts. In each pair, the only difference between the two prompts is the inclusion of Javadoc comments (Section 2.1.2).

Fine-tuning is conducted on a GPU setup with four A100 GPUs (40GB each), utilizing PyTorch’s Accelerate for multi-GPU training. To efficiently train models exceeding 1 billion parameters, we use DeepSpeed [34]. For 7B models (e.g., CodeLlama-7B, CodeGemma-7B), we apply Low-Rank Adaptation (LoRA).

Table 1. Percentage of ground-truth test oracles generated by various model-prompt combinations. Within each prompt pair, we highlight the best-performing prompt in bold.

| LLM \ Prompt Includes: | Test Prefix | Test Prefix, Doc | Test Prefix, MUT Sig | Test Prefix, MUT Sig, Doc | Test Prefix, MUT | Test Prefix, MUT, Doc |
|------------------------|-----------------------------|------------------|----------------------|---------------------------|------------------|-----------------------|
| | | | | | | |
| CodeGPT-110M | 56.80 | 76.16 | 77.28 | 78.28 | 79.83 | 80.07 |
| CodeParrot-110M | 58.96 | 78.93 | 79.58 | 80.23 | 80.48 | 80.88 |
| CodeGen-350M | 58.06 | 78.57 | 79.62 | 80.26 | 80.42 | 81.56 |
| PolyCoder-400M | 57.63 | 77.49 | 78.11 | 78.46 | 79.15 | 80.67 |
| Phi-1 | 58.42 | 78.46 | 80.16 | 79.15 | 79.73 | 80.61 |
| CodeGen-2B | 57.34 | 78.57 | 79.12 | 79.04 | 79.59 | 80.62 |
| CodeGemma-2B | 58.35 | 76.84 | 78.72 | 78.42 | 79.11 | 80.20 |
| PolyCoder-2.7B | 56.04 | 77.60 | 78.65 | 78.28 | 78.90 | 80.23 |
| CodeLlama-7B | 58.38 | 79.22 | 78.90 | 79.07 | 80.75 | 81.61 |
| CodeGemma-7B | 59.53 | 79.25 | 79.66 | 80.41 | 81.08 | 81.84 |
| Average: | 57.95 | 78.11 | 78.98 | 79.16 | 79.90 | 80.83 |
| Std. Dev.: | 0.98 | 0.99 | 0.80 | 0.81 | 0.71 | 0.60 |
| t-test (p-value): | 1.1 * 10 ⁻¹⁹ (s) | | 0.64 (n.s.) | | 0.00798(s) | |

3.1.2 Results. The results are presented in Table 1. Column 1 lists the names of all 10 models, arranged in ascending order by parameter size. Column 2 shows the TOG accuracy when only the Test Prefix is used in the prompt (\mathcal{P}_1), providing a baseline for model performance with minimal contextual information. Column 3 shows the accuracy when Javadoc comments is included in the

| | | |
|---|---|---|
| <pre> ID: 47331 ===== MUT Sig: public boolean accept(File f) ----- * The method allows the display of all directories and all files with the *.db suffix. * @return boolean true if the file should be displayed. ----- //w/ doc.: assertEquals(false, boolean0) // because the file name has no *.db suffix. ////w/ Sig.: assertEquals(true, boolean0); </pre> | <pre> ID: 8267 ===== MUT Sig: public static String valueOf(Object value) ----- * Return a empty String if value is null, else return value * @param value The string value * @return The result String ----- //w/ doc.: assertEquals("", string0); ////w/ Sig.: assertNull(string0); </pre> | <pre> ID: 64241 ===== MUT Sig: public boolean isDoorOrWindow() ----- * Returns always <code>true</code>. ----- //w/ doc.: assertEquals(true, boolean0) ////w/ Sig.: assertEquals(false, boolean0); </pre> |
|---|---|---|

Fig. 3. Examples showing that MUT Sig is not enough to generate correct oracles.

prompt (\mathcal{P}_2). Columns 4 and 5 show the accuracy for prompts \mathcal{P}_3 and \mathcal{P}_4 , both of which incorporate the test prefix and the signature of the MUT. However, \mathcal{P}_4 also includes Javadoc comments. Finally, prompts \mathcal{P}_5 and \mathcal{P}_6 utilize the implementation of the MUT, with \mathcal{P}_6 additionally including Javadoc comments. Rows 12 and 13 show the average accuracy and the standard deviation for each prompt across all models, respectively. The last row shows the t-test results for each prompt pair, indicating whether adding Javadoc comments significantly improved the results.

The average accuracy for \mathcal{P}_1 is 57.95%, and *including Javadoc comments improves the accuracy by 20 percentage points (pp), increasing it to an average of 78.11%*. When examining how TOG accuracy shifts with the inclusion of Javadoc comments ($\mathcal{P}_1 \rightarrow \mathcal{P}_2$), we found that, on average, across CodeParrot-110M, CodeGen-350M, and CodeGemma-7B, 23% of samples shift to match, while only 3.3% shift to no match.

Between \mathcal{P}_2 and \mathcal{P}_3 , we observe that the method signature (MUT Sig) can be as effective as Javadoc comments in TOG. To further investigate, we conduct both qualitative and quantitative analyses of the oracles generated using both prompts.

Theoretically, **Javadoc comments and method signatures serve distinct purposes, though they may contain overlapping information, such as type details and hints about method functionality derived from the MUT name (e.g., getColumnCount()). A method signature provides only structural details—such as input parameters and return types—but lacks the semantic meaning necessary for understanding intent.** For example, a signature may indicate that a method returns a **boolean** value, but it *does not* specify under what conditions it returns true or false. This context—essential for accurate reasoning about a method’s behavior—can *only* be conveyed through a well-written Javadoc comment. Figure 3 presents three examples demonstrating that method signatures *alone* are *insufficient*. In Example (ID: 47331), the MUT signature indicates a boolean return type, but this information is *insufficient* to determine the expected outcome for a given input. In contrast, the corresponding Javadoc comment provides crucial behavioral details: **the method displays all directories and files with the .db suffix and returns true if the file should be displayed.** The input from the test prefix does not have a .db suffix, so it should not be displayed, and thus, the return value should be false. Utilizing this Javadoc comments, the model generated a correct oracle, whereas the MUT signature alone failed to infer the expected outcome. The same pattern holds for the other two examples.

We manually analyzed all cases where Javadoc comments enabled the generation of ground truth test oracles, whereas the MUT signature failed to do so. Our analysis reveals that Javadoc comments play a crucial role in defining method behavior: in 51.8% of cases, they explicitly specify

the method's input and expected output, establishing a clear contract (**input/output**). In 22.8%, they emphasize what the method returns, helping in understanding the *expected* output (**return**). Additionally, 15.6% describe the method's overall functionality without detailing specific inputs or outputs (**functionality**), while 9.6% provide general descriptions that lack focus on return or input/output behavior (**description**). These findings highlight that **well-written Javadoc comments provide crucial information for TOG that MUT signature alone fail to convey**.

While our RQ1 study filtered out samples with **empty Javadoc comments**, random sampling revealed significant variation in their length and quality across the dataset. Notably, 9% of Javadoc comments associated with mismatched oracles were shorter than 42 characters, including vague/irrelevant documentation such as `/* Bean Method */`, `/ @return */`, `/* Default ctor */`, `/* @InheritDoc */`, and `/* Document me */`. Because the models fine-tuned on \mathcal{P}_2 were trained to focus on Javadoc comments, poor-quality comments negatively impacted accuracy (more discussion in RQ5). However, this only indicates that **developer-written Javadoc comments are not always well-constructed—not that Javadoc comments themselves are ineffective**.

We further analyzed 20 randomly selected cases where Javadoc comments *did not* match the ground truth, but the method signature did. After carefully reviewing the test prefix, MUT, and Javadoc comment, we categorized the results into five cases: (**Case 1**) 7 oracles were *incorrect*, (**Case 2**) 3 were *correct and equivalent* to the ground truth, (**Case 3**) 2 were *correct but weaker* than the ground truth, (**Case 4**) 1 was *correct and stronger* than the ground truth, and (**Case 5**) 7 were *correct but incomparable* to the ground truth. **An example of case 4 (ID: 22119) involves a Javadoc comment stating, "This accessor method returns a reference to the live list, not a snapshot". The ground truth oracle is `assertNotNull(list1)`, whereas the oracle generated from the Javadoc comments was `assertSame(list1, list0)`, where both lists are derived from calls to the MUT. Here, the LLM leveraged the semantics of "live list" and "not a snapshot" to infer that the MUT is **idempotent**, leading to a **stronger** test oracle. Overall, among these 20 oracles generated using Javadoc comments that did not match the ground truth, 13 were correct and capable of effective bug detection. In RQ4, we conducted a bug detection study using real-world bugs, and our results show that oracles generated with only Javadoc comments can be even more effective than those based on the entire MUT implementation in detecting bugs.**

Between prompts \mathcal{P}_3 and \mathcal{P}_4 , including Javadoc comments also improves accuracy, with a similar trend observed between \mathcal{P}_5 and \mathcal{P}_6 (p-values for the t-test are shown in the last row of Table 1). **Notably, minimal information—specifically, prompt \mathcal{P}_2 —yields performance close to that of the maximum-information prompt \mathcal{P}_6 .** While leveraging MUT code can benefit TOG, it also has inherent limitations: models can inadvertently *learn buggy behaviors from buggy code*, resulting in false negative test oracles that fail to detect bugs, as illustrated in Figure 1. **Findings from this study demonstrate that using only Javadoc comments—without MUT—can still achieve nearly the same performance.** The MUT and Javadoc serve as *independent sources of program behavior*, and using them in separate prompts enables independent oracle generation, which can *help identify inconsistencies or conflicts between the MUT and Javadoc*, as shown in Figure 1.

Table 2. Percentage of ground-truth oracles generated with different versions of Javadoc comments.

| Accuracy LLM | Original Javadoc | Removed Desc. | Removed @param | Removed @return | Removed @throws | Removed @see | Removed Desc. + @return |
|-----------------|---------------------|------------------|-------------------|--------------------|--------------------|-----------------|-------------------------------|
| CodeGPT-110M | 78.27 | 68.56 | 77.26 | 72.80 | 78.23 | 77.91 | 54.6 |
| CodeParrot-110M | 80.39 | 70.53 | 80.0 | 74.78 | 80.46 | 80.0 | 55.82 |
| CodeGen-350M | 78.48 | 69.17 | 77.87 | 73.66 | 78.59 | 77.94 | 55.21 |
| PolyCoder-400M | 77.37 | 66.76 | 77.12 | 70.75 | 77.23 | 76.72 | 52.66 |
| Phi-1-1.3B | 78.38 | 67.37 | 78.12 | 72.80 | 78.38 | 77.84 | 53.63 |
| CodeGen-2B | 78.34 | 68.95 | 78.09 | 73.09 | 78.30 | 77.8 | 53.95 |
| CodeGemma-2B | 76.65 | 68.30 | 76.43 | 72.19 | 76.40 | 76.33 | 53.48 |
| PolyCoder-2.7B | 77.51 | 67.15 | 76.90 | 72.58 | 77.33 | 76.9 | 52.98 |
| CodeLlama-7B | 79.13 | 67.41 | 78.20 | 73.05 | 79.24 | 78.66 | 54.78 |
| CodeGemma-7B | 79.17 | 67.05 | 78.59 | 73.52 | 79.20 | 78.63 | 55.0 |
| Avg: | 78.37 | 68.12 | 77.86 | 72.92 | 78.34 | 77.87 | 54.12 |

RQ1 Finding: Including Javadoc comments across all three prompt pairs provides additional context about the MUT’s behavior, improving TOG to better align with ground truth oracles. Notably, using Javadoc alone (\mathcal{P}_2) achieves performance comparable to the maximum-information prompt (\mathcal{P}_6). In 74% of cases where Javadoc outperformed the MUT signature, Javadoc comments contained substantive input/output and return value details, highlighting the impact of documentation quality on TOG effectiveness. For non-matching Javadoc-generated oracles, 65% were still correct and capable of detecting bugs, suggesting that an ‘exact match’ with ground truth oracle may underestimate the value of Javadoc comments.

3.2 RQ2: Impact of Different Javadoc Components on TOG

A Javadoc comment can include various components, such as a general description of the MUT, parameter, return, throws, and see tags, along with metadata like version and author details. These comments can be lengthy, and since LLMs have token limits, a key question arises: which **components** contribute most to improving test oracle generation?

Beyond token limits, writing detailed Javadoc comments can be time-consuming for developers. Identifying the most impactful components can help streamline documentation efforts while maximizing automated TOG effectiveness.

To address this, in RQ2, we systematically remove different Javadoc components to assess their impact on TOG accuracy.

3.2.1 Experimental Setup. We identified the most frequent Javadoc components—description, @param, @return, @throws, and @see—and created six modified versions of the Javadoc comments by removing them individually and removing the description and @return together. We then performed inference using prompt \mathcal{P}_2 with the test prefix and modified Javadoc comments to assess their impact on TOG. This experimental design allowed us to study the impacts of the qualitative observations from RQ1 in a more controlled fashion.

3.2.2 Results. The results are presented in Table 2. With the original Javadoc comments, the average TOG accuracy is 78.37%. Removing the description has the most significant impact, reducing accuracy to 68.12%—a 10 percentage points (pp) drop. We manually investigated the 295 affected samples and found that *the description often contains unique information about the MUT that other Javadoc comment tags do not cover, directly impacting the accuracy of generated oracles.*

```

//Example: 1
=====
/**
 * Returns a string
 * representation of this type.
 */
-----
public void test7() {
    Type t0 =
        Type.getObjectType("[B]");
}
-----
//w/ desc.: assertEquals("[B]",
    t0.toString());
////w/o desc.:
    assertEquals("[B]",
        t0.getDescriptor())

//Example: 2
=====
/**
 * Method getColumnCount
 */
-----
public void test3() {
    PrimaryKey pKey = new
        PrimaryKey();
    Column c0 = new Column();
    pKey.addColumn(c0);
    boolean bool0 = pKey0.isValid();
}
-----
//w/ desc.: assertEquals(1,
    pKey.getColumnCount());
////w/o desc.: assertEquals(true,
    bool0);

//Example: 3
=====
/**
 * Get the time in milliseconds
 * ...
 * Default value is 1000 (1 second);
 */
-----
public void test9() {
    FBEventManager fM = new
        FBEventManager();
    long l0 = fM.getWaitTimeout();
}
-----
//w/ desc.: assertEquals(1000L,
    l0);
////w/o desc.: assertEquals(0L,
    l0);

```

Fig. 4. Impact of Javadoc comment's *description* on test oracle generation

```

//Example: 1
=====
/**
 * Get the L Norm used.
 * @return the L-norm used
 */
-----
public void test1() {
    NaiveBayesMultinomialText nBText = new
        NaiveBayesMultinomialText();
    String s0 = nBText.stopwordsTipText();
}
-----
//w/ @return: assertEquals(2.0,
    nBText.getLNorm(), 0.01D);
////w/o @return: assertEquals(2.0,
    nBText.getLNorm(), 0.01D);

//Example: 2
=====
/**
 * @return Returns the selectFetchSize.
 */
-----
public void test14() {
    DBCopyPreferenceBean pBean = new
        DBCopyPreferenceBean();
    pBean.setUseFileCaching(false);
}
-----
//w/ @return: assertEquals(1000,
    pBean.getSelectFetchSize());
////w/o @return: assertNull(pBean.getName());

```

Fig. 5. Examples where removing @return tag affects (right) and does not affect (left) the generated oracle.

Figure 4 presents three examples, each showing the original Javadoc comments (light blue), the test prefix, and assertions oracles generated *with* (green) and *without* (red) the description. In each case, the description clearly outlines the MUT's behavior and provides hints for *what to check* and *what to check against*. For instance, in Example 3, the description specifies that the oracle should check the *wait timeout* and default expected value should be 1000 milliseconds.

When the @return tags are removed, the average accuracy decreases by 5.45 pp. We *observed a significant overlap between the description and the @return tag in the Javadoc comments, which helps explain why removing all @return tags results in only a 5.45 pp drop in accuracy*. Figure 5 presents two examples: one where removing the @return tag *affects* the generated oracle and one where it *does not*. For example 1 (left), even though we remove "@return the L-norm used" portion from the Javadoc comments, the description still conveys this information. As a result, removing the @return tag does not impact the generated assertion. Conversely, in example 2 (right), removing the tag directly affects the oracle, as no other part of the Javadoc comments provides this information. @return tags provide key hints, specifying *what to check*, *what to check against*, and *their types*. In Figure 5 (Examples 1 and 2), they guide the verification of L-norm and selectFetchSize, directly helping in assertion oracles generation.

```

//Example: 1
=====
"/**
 * @param x The horizontal position of the
 *           center of the symbol.
 * @param y The vertical position of the
 *           center of the symbol.
 */"
-----
public void test0() {
    BoxURSymbol symbol0 = new BoxURSymbol();
    symbol0.generatePoints((-2016), (-2016));
    symbol0.generatePoints((-1209), 198);
}
-----
//w/ @param: assertEquals(8, symbol0.getSize());
///w/o @param: exception

//Example: 2
=====
"/**
 * @see de.outstare.fortbattleplayer.
 *           model.Sector#getHeight()
 */"
-----
public void test20() {
    CharacterClass c0 = CharacterClass.SOLDIER;
    SimpleSector s0 = new SimpleSector(1838,
        true, true, 1838, 1838, true, 1838,
        c0);
    boolean boolean0 = s0.equals((Object) null);
}
-----
//w/ @see: assertEquals(1838, s0.getHeight());
///w/o @see: assertEquals(false, boolean0);

```

Fig. 6. Impact of removing @param tag (left) and @see tag (right) on test oracle generation.

To further investigate this overlap, we removed both the description and the @return tags, affecting 10% of the samples (288). Removing the description alone caused a 10 pp accuracy drop, while removing the @return tags alone resulted in a 5 pp drop. Combined, this led to a 25 pp accuracy drop, as shown in Column 8 of Table 2. **In nearly all 288 affected cases, the description and return tag contained the same information, providing the same hints for oracle generation.**

With the removal of the @param, @throws, and @see tags, we observed that the *accuracy dropped by less than 1 pp on average*. In examining the 27 samples where removing the @param tag caused the generated oracle to *differ* from the ground truth oracle, we found that the ground truth oracles were more verbose, using fully qualified variable names within the assertion predicates. These fully qualified names were often learned from the @param tag description. *Although these 27 samples did not match the ground truth, the oracles generated without the @param tag were generally more concise*. In Figure 6, we present two examples where the @param and @see tags provided hints in generating assertion oracles that aligned with the ground truth oracles.

Regarding the @throws tag removal, *we observed that for 4 out of 10 LLMs, accuracy actually improved slightly, though this improvement was negligible*. In reviewing some samples, we found that when the @throws tag specified a null pointer exception for null values, the generated assertion oracles often check not null condition using a `assertNotNull()` oracle. While these oracles did not exactly match the ground truth, they were still correct.

RQ2 Finding: The *description* and *@return* tags provide the richest behavioral information about the MUT and are most valuable for automated TOG. These components also significantly overlap. In contrast, @param, @throws, and @see tags contribute little to TOG and can be omitted without significant accuracy loss.

3.3 RQ3: Assessing GPT-Generated Javadoc Comments

In RQ1 and RQ2, we investigate the impact of Javadoc comments and their individual components on TOG, relying on developer-written or derived documentation. However, source code often lacks sufficient documentation—56% of the SF110 dataset, for example, lacks method-level comments. This raises the question of *whether GPT-generated Javadoc comments, which summarize MUT behavior, can similarly support TOG*. To explore this, we generate Javadoc comments directly from the method implementation using a GPT model and evaluate their impact on TOG accuracy.

```

// Example: 1
=====
---GPT-generated Javadoc
/**
 * Returns whether the object is coded or not.
 * @return true if the object is coded, false otherwise
 */
-----
public void test14() {
    RawVariable rV = new RawVariable();
    SupportingDocument sDoc =
        new SupportingDocument();
    boolean b0 =
        rV.containsSupportingDocument(sDoc);
}
-----
// Same as ground truth assertion
// w/ GPT-doc: assertEquals(false, rV.isCoded());
////w/o GPT-doc: assertEquals(false, rV.isCleaned());

// Example: 2
=====
---GPT-generated Javadoc
/**
 * This method retrieves the current index value.
 * @return The current index value.
 */
-----
public void test0() {
    Object obj = new Object();
    NoteListDataEvent event = new NoteListDataEvent(obj,
        2);
    int type = event.getType();
}
-----
// Same as ground truth assertion
// w/ GPT-doc: assertEquals(2, event.getIndex());
////w/o GPT-doc: assertEquals(2, type);

```

Fig. 7. Impact of GPT-generated Javadoc comments on ground truth oracle generation.

3.3.1 Experimental Setup. For the test portion of the SF110* dataset, consisting of 2,780 input samples, we generate Javadoc comments using **OpenAI’s gpt-3.5-turbo-0125** model. The generation prompt can be found in our project repository.

Table 3. Percentage of ground-truth test oracles generated with GPT-generated Javadoc comments.

| Model Prompt | CodeGPT 110M | CodeParrot 110M | CodeGen 350M | PolyCoder 400M | Phi 1.3B | CodeGen 2B | CodeGemma 2B | PolyCoder 2.7B | CodeLlama 7B | CodeGemma 7B | Avg. |
|---|-----------------|--------------------|-----------------|-------------------|-------------|---------------|-----------------|-------------------|-----------------|-----------------|--------------|
| \mathcal{P}_1 | 56.80 | 58.96 | 58.06 | 57.63 | 58.42 | 57.34 | 58.35 | 56.04 | 58.38 | 59.53 | 57.95 |
| \mathcal{P}_1 + GPT-generated Javadoc | 66.07 | 70.79 | 69.71 | 64.42 | 67.73 | 70.54 | 67.30 | 67.87 | 66.90 | 66.72 | 67.80 |

3.3.2 Results. In Table 3, we present the TOG performance of 10 different fine-tuned models on two distinct prompts. The first prompt (\mathcal{P}_1) includes only the test prefix, while the second prompt (\mathcal{P}_1 + GPT-generated Javadoc) incorporates both the test prefix and the generated Javadoc comments. The purpose of using these two prompts is to isolate the impact of GPT-generated Javadoc comments on TOG. When using \mathcal{P}_1 , the average accuracy is 57.95%. *Incorporating GPT-generated Javadoc comments increases the average accuracy to 67.8%, indicating a 10 pp improvement.*

From the samples showing improvement (from no match to match) after adding generated Javadoc comments, we manually investigated 20 cases. When comparing the test oracles generated using \mathcal{P}_1 (test prefix only) to those generated with \mathcal{P}_1 + GPT-generated Javadoc, we observed that the GPT Javadoc provided useful hints that were instrumental in constructing the oracle statements. Figure 7 illustrates two representative examples of generated Javadoc comments (light blue), the assertions generated using only the test prefix (red), and the assertions generated using both the test prefix and the GPT-generated Javadoc comments (green). Each Javadoc comment includes a summary of the MUT and a @return tag, explicitly specifying the method’s return behavior. In RQ2, we established that the method summary and @return tag are the most impactful components for automated test oracle generation. These examples further reinforce this finding.

RQ3 Finding: We find that including GPT-generated Javadoc comments with the test prefix improves TOG accuracy by 10 pp. This generalizes the finding in RQ1 that Javadoc comment provides additional context about MUT behavior, enhancing the generation of effective test oracles. Moreover, GPT-generated Javadoc comments that contributed to generating ground truth oracles consistently include a method description and a @return tag, further supporting the RQ2 finding that these components are the most valuable for TOG.

3.4 RQ4: Impact of Javadoc Comments on Real Bug Detection

In the previous RQs, we examined the correctness of automatically generated test oracles. While *correctness* is a critical property [16], the *ability to detect bugs*—referred to as the *strength property*—is equally important. In this research question, we investigate the impact of Javadoc comments on bug detection using *Defects4J*, a widely adopted real-world Java bug benchmark. We evaluate multiple fine-tuned LLMs and prompts to compare the relative strengths of the generated oracles and benchmark our results against two state-of-the-art methods.

Table 4. Defects4J bug detection performance of model-prompt pairs on the dataset subset (288/374), where each input sample *includes* a Javadoc comment.

| Model \ Prompt | CodeParrot-110M | CodeGen-350M | Phi-1.3B | CodeLlama-7B | CodeGemma-7B |
|---|-----------------|--------------|----------|--------------|--------------|
| prefix (\mathcal{P}_1) | 44 (4) | 44 (4) | 42 (4) | 42 (5) | 45 (4) |
| prefix + Javadoc comments (\mathcal{P}_2) | 44 (4) | 50 (10) | 42 (4) | 48 (11) | 48 (7) |
| Total Bugs (Unique) : | 48 | 54 | 46 | 53 | 52 |

3.4.1 Experimental Setup. Details of the Defects4J dataset are provided in Section 2.2.1. Similar to prior works [10, 11, 17], this dataset includes a total of 374 input samples. Each sample comprises a test prefix, a method under test, and its corresponding Javadoc comments. For the 86 samples where Javadoc comments were *unavailable*, we generated them using the procedure described in RQ3 (Section 3.3).

Based on RQ1 results, we selected the top five models that performed best on prompt \mathcal{P}_2 while ensuring diversity by avoiding repetition within the same model family: **CodeParrot-110M, CodeGen-350M, Phi-1.3B, CodeLlama-7B, and CodeGemma-7B.**

After an initial study on all five models using \mathcal{P}_1 and \mathcal{P}_2 on the Defects4J dataset (Table 4), we selected the top three for further analysis: **CodeGen-350M, CodeLlama-7B, and CodeGemma-7B.** Using these models and prompt pairs ($\mathcal{P}_1, \mathcal{P}_2$) and ($\mathcal{P}_5, \mathcal{P}_6$), we generated test oracles to detect Defects4J bugs. Table 5 reports the total number of unique bugs detected by different model-prompt combinations.

3.4.2 Results. Table 4 presents the Defects4J bug detection performance of five models using prompts \mathcal{P}_1 and \mathcal{P}_2 . Here, \mathcal{P}_1 includes only the test prefix, while \mathcal{P}_2 includes both the test prefix and Javadoc comments. This study evaluates 288 input samples, all containing Javadoc comments. Most models generated stronger test oracles and detected more bugs with \mathcal{P}_2 than with \mathcal{P}_1 . This *enhanced bug detection performance can be directly attributed to the Javadoc comments*, as they are the only distinguishing feature between these two prompts.

For example, CodeGen-350M, CodeLlama-7B, and CodeGemma-7B detected more bugs with \mathcal{P}_2 , highlighting the value of Javadoc comments in generating stronger test oracles. CodeGen-350M

Table 5. Defects4J bug detection performance of various model-prompt pairs

| Prompt \ Model | CodeGen-350M | CodeLlama-7B | CodeGemma-7B |
|---|---------------------|---------------------|---------------------|
| prefix (\mathcal{P}_1) | 59 (5) | 58 (8) | 62 (3) |
| prefix + Javadoc comments (\mathcal{P}_2) | 67 (13) | 63 (13) | 68 (9) |
| Total Unique ($\mathcal{P}_1 + \mathcal{P}_2$): | 72 | 71 | 71 |
| prefix + MUT (\mathcal{P}_5) | 62 (4) | 68 (5) | 62 (4) |
| prefix + MUT + Javadoc comments (\mathcal{P}_6) | 63 (5) | 73 (10) | 65 (7) |
| Total Unique ($\mathcal{P}_5 + \mathcal{P}_6$): | 67 | 78 | 69 |

and CodeLlama-7B detected 10 and 11 unique bugs, respectively, using Javadoc comments. The last row shows the total number of unique bugs detected when combining \mathcal{P}_1 and \mathcal{P}_2 .

For CodeParrot-110M and Phi-1.3B, the inclusion of Javadoc comments does not improve performance, suggesting that different fine-tuned models generalize differently during unseen inference. CodeParrot-110M, the smallest model in this study, and Phi-1.3B, pre-trained on Python datasets, may have limited generalization capabilities for Java code generation. Despite performing well in RQ1, these models struggle to generalize effectively in the TOG task. Based on these results, we select the top three models—CodeGen-350M, CodeLlama-7B, and CodeGemma-7B—for deeper investigation.

Table 5 presents the bug detection results for the top three models across all 374 input samples. This study evaluates two prompt pairs, totaling four distinct prompts. Notably, including all 374 samples in this evaluation resulted in a higher total bug detection count for both prompts compared to Table 4.

Row 2 presents bugs detected using \mathcal{P}_1 (test prefix), while Row 3 shows results for \mathcal{P}_2 (test prefix + Javadoc comments). With Javadoc comments, CodeGen-350M, CodeLlama-7B, and CodeGemma-7B detected 8, 5, and 6 additional bugs, respectively. A similar trend is observed between \mathcal{P}_5 and \mathcal{P}_6 . In summary, Javadoc comments, as a *complementary* source of information, consistently led to the detection of more Defects4J bugs.

Comparing results from \mathcal{P}_2 and \mathcal{P}_5 , we find that *Javadoc comments alone can encode sufficient contextual information to replace the MUT code entirely, achieving comparable or even superior performance in TOG and bug detection*. \mathcal{P}_2 (prefix + Javadoc comments) detects a maximum of 68 and an average of 66 bugs, while \mathcal{P}_5 (prefix + MUT) detects a maximum of 68 and an average of 64 bugs. *This suggests that Javadoc comments can be used in place of MUT implementation for TOG.*

We also find that for 2 of the 3 models, using MUT implementation does not improve bug detection compared to Javadoc comments (\mathcal{P}_2 vs. \mathcal{P}_6). A possible reason is that models may struggle to extract high-level behavioral insights from complex MUT code, while Javadoc comments offer a more accessible, natural-language summary.

In summary, Javadoc comments, when used as a standalone input, achieved comparable or better performance than MUT code. Thus, they have the potential to replace MUT while maintaining similar performance. The impact of Javadoc comments remains significant even when the code for the MUT is included. When Javadoc comments are used with the prefix and MUT (Row 6), the total number of detected bugs increases in all cases. Notably, the best performance is achieved by CodeLlama-7B, which detected a total of 73 bugs. Furthermore, the total number of unique bugs is

Table 6. Impact of Javadoc comments on Defects4J bug detection.

| Bug ID | Cli, 34 | Csv, 6 |
|-------------------|---|--|
| Bug Details | <p>getParsedOptionValue returns null unless Option.type gets explicitly set.</p> <p>The user expects it to be String unless set to any other type.</p> <p>This could be either fixed in the Option constructor or in CommandLine.getParsedOptionValue. Mentioning this behaviour in Javadoc would be advisable.</p> | <p>if .toMap() is called on a record that has fewer fields than we have header columns we will get an ArrayOutOfBoundsException.</p> <p>CSVRecord.toMap() fails if row length shorter than header length.</p> |
| Javadoc Comment: | <pre>/** * Retrieve the type of this Option. */</pre> | <pre>/** * Returns the number of this record in the * parsed CSV file. */</pre> |
| Test Prefix: | <pre>public void test09() { Option op0 = new Option(" ", " ", true, " "); Object obj0 = op0.getType(); }</pre> | <pre>public void test17() { HashMap<String, Integer> hMap = new HashMap<String, Integer>(); Integer int0 = new Integer(854); hMap.put((String) null, int0); String[] strA0 = new String[0]; CSVRecord cSVRec = new CSVRecord(strA0, hMap, (String) null, 854); cSVRec.toMap(); }</pre> |
| Javadoc not used: | <pre>assertEquals(true, op0.isIsSet());</pre> | <pre>assertEquals(854, cSVRec.size())</pre> |
| Javadoc used: | <pre>assertNotNull(obj0);</pre> | <pre>assertEquals(854L, cSVRec.getRecordNumber());</pre> |

substantially higher when Javadoc comments are included, highlighting their critical role in TOG and bug detection.

Figure 6 presents two Defects4J bugs uniquely detected due to Javadoc comments. For example, bug 34 in the *Cli* project occurs when the `Option` type is not explicitly set. If undefined, the user expects it to default to a `String`. Without Javadoc comments, the generated test oracle passed in both fixed and buggy versions, failing to detect the bug. However, with Javadoc comments, the assertion ensured the type was not null, passing in the fixed version but failing in the buggy one—successfully detecting the bug. This demonstrates how Javadoc comments strengthen test oracles by capturing intended behavior not directly inferred from the MUT.

Similarly, in bug 6 of the *Csv* project, `CSVRecord.toMap()` fails when the row length is shorter than the header length, triggering an `ArrayOutOfBoundsException`. Without Javadoc comments, the generated oracle only checks the `CSVRecord` size, producing an incorrect test that failed to detect the bug. In contrast, with Javadoc comments, the oracle checks the record number as specified, successfully detecting the bug.

Comparison With Baselines: We compare our results with two baseline methods. The first is *TOGA* [10], a neural approach for test oracle generation. *TOGA* utilizes the test prefix, MUT, and Javadoc comments to generate test oracles for detecting Defects4J bugs. While *TOGA* detected a total of 57 bugs, a significant portion of these were identified using implicit oracles. Specifically, when *TOGA* failed to generate explicit assertion oracles, it executed the test prefix (not generated by *TOGA*) on buggy programs, where uncaught exceptions caused test failures, indirectly indicating bug detection. In contrast, our method does not execute test prefixes when no oracles are generated.

Despite this, our approach demonstrates a significant improvement over TOGA, detecting 73 bugs when the same input information is used and 68 bugs even when MUT code is excluded from the input prompt.

We note that using the MUT from the fixed version, as TOGA does, is impractical because fixed versions are typically unavailable in real-world scenarios, as demonstrated by prior research [18, 26]. Moreover, if the buggy MUT is used, the generated oracles risk learning from buggy behavior, which increases the likelihood of producing regression oracles that fail to detect bugs in that program version. Therefore, achieving state-of-the-art bug detection capability using only Javadoc comments, as demonstrated by our method, offers a viable solution to mitigate these challenges.

The second baseline method is *nl2postcondition* [11], which uses GPT-4 and StarChat to generate test oracles from Javadoc comments and buggy MUT code. This method also incorporates class-level in-file comments, *which are not utilized in our study*. For each input sample (comprising Javadoc comments and buggy MUT code), *nl2postcondition* generates 10 assertions. If at least one of the generated assertions passes on the fixed version and fails on the buggy version, the method reports a detected bug. In contrast, our study generates only a single test oracle per input sample.

Despite additional context and advantages, *nl2postcondition*—using GPT-4 with Javadoc comments and buggy MUT code—detected a total of **47 unique bugs** [11]. In contrast, our method, relying solely on Javadoc comments and generating a single assertion per sample, detected 68 unique bugs—an improvement of nearly 45%. When the MUT was removed from *nl2postcondition*, its performance dropped significantly, detecting only 35 bugs. This highlights a 94% improvement by our method, which detected 68 bugs under the same conditions.

The significant improvement of our approach can be attributed to fine-tuning with a high-quality dataset collected from diverse, real-world projects, well-designed prompts, and careful selection of the most relevant parts of the Javadoc comments, as determined by our RQ2 findings.

RQ4 Finding: The most important finding is that Javadoc comments alone can encode sufficient contextual information to potentially replace the MUT code, achieving comparable or even better performance in TOG and bug detection. Across all models, prompts with Javadoc comments detect more bugs, indicating that the generated oracles are stronger and better at capturing developer intentions. Our method, without using the MUT, detects 19–45% more bugs than prior work that uses the MUT, and 94% more bugs when prior methods exclude the MUT.

3.5 RQ5: Helpful or Harmful Aspects of Javadoc Comments

In this research question, we conduct a comprehensive qualitative and quantitative analysis to identify the aspects of Javadoc comments that are *helpful* or *harmful* for test oracle generation. First, we present a taxonomy categorizing the reasons why Javadoc can enhance or hinder test oracle generation. Then, we provide concrete examples of cases where adding Javadoc comments did not improve performance and analyze the underlying reasons.

3.5.1 Experimental Setup. To identify **positive factors**, we extracted cases where Javadoc comments helped generate a ground truth oracle. For **negative factors**, we examined instances where Javadoc comments resulted in oracles that did not match the ground truth. To identify cases where Javadoc comments had **no impact**, we analyzed situations where the generated oracle remained **unchanged**. Our prompt design naturally supports these investigations.

We conducted a manual investigation to develop a taxonomy of why Javadoc comments are helpful or harmful for test oracle generation. This involved systematically analyzing Javadoc

| | | |
|---|---|--|
| <pre> ID: 54958 (Return behavior) ===== /** * Returns the single line comment * start string. * @return the start string */ ----- //w/ doc.: assertEquals("//\"", doc.getSingleLineCommentStart()); ///w/o doc: assertEquals(2, doc.getIndentationSize()); </pre> | <pre> ID: 33999 (Input/Output) ===== /** * Removes all null values from the * given string array. * Returns a new string array that * contains all none null values * of the input array. * @param strings The array to be * cleared of null values */ ----- //w/ doc: assertNull(stringArray0); ///w/o doc.: assertNotNull(stringArray0); </pre> | <pre> ID: 141507 (Functionality) ===== /** * Returns the algorithm for the * QuickServer used for key * management * when run in a secure mode. * @see #setAlgorithm */ ----- //w/ doc.: assertEquals("{_dWSu0rBmFf", secureStore0.getAlgorithm()); ///w/o doc.: something else </pre> |
|---|---|--|

Fig. 8. Examples showing positive reasons of Javadoc comments for effective TOG

comments, identifying recurring patterns, and categorizing the reasons. We iteratively refined these categories to ensure clear and meaningful distinctions.

3.5.2 Results. To find the positive reasons we have manually investigated a total of **900** samples across all prompt pairs ($\mathcal{P}_1 \rightarrow \mathcal{P}_2, \mathcal{P}_3 \rightarrow \mathcal{P}_4, \mathcal{P}_5 \rightarrow \mathcal{P}_6$). We identified **four** positive reasons why adding Javadoc comments helped generate a ground truth test oracle:

Return behavior When the Javadoc comments explicitly defines the **return value**, either in the **description (using "Returns...")** or via the `@return` tag.

Functionality When the Javadoc comments clearly states the **purpose** or **expected behavior** of the method.

Input/output When both method **parameters** and **return** values are *explicitly* stated, often including the `@throws` tag.

Description When the Javadoc comments provides some descriptive information, though with **limited** detail.

Figure 8 presents three examples. In **ID: 54958**, we show how the return behavior described in the Javadoc comments directly influenced ground truth oracle generation. In this category, Javadoc comments also include additional details such as *method functionality* or *general descriptions*, but the return behavior played the most significant role in guiding oracle generation. As a result, we classified these cases under **Return behavior**. *Return behavior documentation provides critical hints for both "what to check" and "what to compare with"*. For instance, in **ID: 54958**, the Javadoc comment specifies that "what to compare with" is the "single line comment". Similarly, the examples in Figure 5 illustrate "what to check". Notably, this category accounts for **65.64% of all positive contributions**, emphasizing the crucial role of well-defined return behavior for TOG.

In example **ID: 141507**, the Javadoc comment states that the method returns the algorithm for the QuickServer used for key management when running in secure mode and references `@see #setAlgorithm`. This information helped construct the oracle by guiding the model to learn "what to compare with" from the test prefix and "what to check" from the Javadoc comment. This **Functionality** category accounts for 26.93% of all positive contributions.

In example **ID: 33999**, the Javadoc comment describes the input and output behavior of the MUT, aiding in the generation of the ground truth oracle. Since the test prefix provides the input, the model inferred the expected output from the Javadoc comments. This **Input/Output** category accounts for 3.05% of all positive contributions.

The last category, **Description** only provides very limited context on the MUT, e.g., 'This method computes the dot product', or 'Helper method for encoding an array of bytes

as a Base64 String' with no other information. In the 4.36% of the cases in our study, the specifics of these descriptions did help the TOG task – *perhaps because the LLMs have learned a good embedding of the mentioned concepts, e.g., 'dot product'.*

Solving the test oracle problem fundamentally requires understanding the expected behavior of a method for a given input. Therefore, as long as Javadoc comments convey this information, they contribute to effective oracle generation. While we have categorized the reasons, they ultimately serve the same core purpose—providing guidance on *what to check* and *what to expect*.

To identify the **negative** characteristics of Javadoc comments, we analyzed all 229 samples in which the first prompts in a pair, e.g., \mathcal{P}_3 , matched the ground truth but the second, e.g., \mathcal{P}_4 , did not. Among these, 37.1% were **correct** oracles that did not match the ground truth. Of the remaining 144 samples, we identified **three** distinct reasons why Javadoc comments are **not helpful**:

Lengthy: When the Javadoc comment was excessively long it could exceed the LLM token limit; this occurred in 3.5% of the incorrect samples. IDs 100041, 2322, 100332, and 780 all have very detailed input/output descriptions, but the length of prefix+signature+doc **exceeds the token length**, so no oracle is generated.

Vague/Incomplete: When the Javadoc comment contained some **relevant** information but **did not explicitly describe all input/output behavior or return values**; this occurred in 55.6% of the incorrect samples. We describe four randomly selected cases: ID 44613 has a description of `/* isophote flow??? */`; ID 49847 has a description of `/* Method getXMLName */`, **which adds nothing beyond part of the information in the signature**; ID 156278 has a description of `/* toString means print the data string, unless the data has not been read at all. */`, **which does not define either input/output behavior or the functionality of the MUT**; and ID 135723 has a description of `/* @inheritDoc*/`.

Irrelevant: When the Javadoc comment included text **unrelated to the MUT**; this occurred in 15.3% of the incorrect samples. We describe four randomly selected cases: ID 10004 has a description that references various methods and classes but ends with `@return Document me!`; IDs 9790 and 11287 have Javadoc containing a meaningful `@return` tag, but the description consists only of `DOCUMENT ME!`; and ID 16507 has the following description:

```
/* Taken from Eammon McManus' blog:
 * http://weblogs.java.net/blog/emcmanus/archive/2007/03/getting_rid_of.html.
 * Prevents the need for placing SuppressWarnings throughout the code. */
```

We categorized samples as **unknown** when none of the above categories applied; this occurred in 25.7% of the incorrect samples.

We further analyzed cases where adding Javadoc comments **did not change the results**. From $\mathcal{P}_1 \rightarrow \mathcal{P}_2$, we identified 512 instances where the generated oracles **remained unchanged (non match with ground truth)**. Among them, 196 had identical oracles for both prompts. **Upon analysis, we found that 88 were correct, 66 were vague, 22 were irrelevant, 17 were unknown, and 3 were lengthy.** These findings align with the previously discussed **negative reasons**.

In summary, Javadoc comments can contain excessively long text, vague/insufficient details (e.g., `@inheritDoc`, `DOCUMENT ME!`), or consist solely of a `@see` tag. Additionally, many fail to describe the input/output contract or specify return values. Due to these shortcomings, such Javadoc comments do not contribute to effective oracle generation. **We conjecture that lengthy, vague/incomplete, or irrelevant documentation misguides the model, preventing it from learning meaningful behavioral constraints.** Our findings emphasize the need for well-structured documentation, guiding developers to improve Javadoc comments for effective TOG.

4 THREATS TO VALIDITY

In our large-scale study, we leveraged the widely recognized and open-source SF110 dataset [13]. However, the results from this dataset may not generalize when using other datasets. To investigate the generalizability, we additionally perform a study on the unseen Defects4J dataset [23].

We developed several tools and scripts to automate various steps and data analysis. While there is a possibility that our implementation may contain bugs, we took several precautions to minimize this risk. Each experiment was repeated at least three times to ensure result consistency. Furthermore, we utilized publicly available libraries to reduce the likelihood of errors and performed comprehensive validity checks throughout the process.

Our qualitative evaluation involved interpreting Javadoc comments by the authors. To mitigate subjectivity, each author independently analyzed samples to identify helpful and harmful characteristics. We then integrated these findings to define the seven characteristics presented in RQ5 and used them for the larger qualitative analysis. While we did not analyze each sample by multiple authors, we cross-checked a random subset of each other's analyses to ensure consistency.

In our investigation, we utilized the *exact match* metric during fine-tuning to compute oracle generation accuracy, a widely adopted metric in the research community. However, we acknowledge that this metric may underestimate accuracy, as correct test oracles can take many forms. For unseen inference on the Defects4J dataset, we relied on test validation. We assert that this methodology offers a reliable accuracy metric, as it is sensitive *only* to the correctness of the generated oracles, rather than their *syntactic similarity*.

5 RELATED WORKS

5.1 Automated Test Oracle Generation

Automated test oracle generation (TOG) methods often employ a combination of approaches, broadly categorized into four types: fuzzing-based, search-based, specification mining-based, and machine learning-based. The fuzzing-based approach includes [31, 45]. Most fuzzing-based approaches rely on implicit oracles such as exceptions thrown due to unintended behavior (e.g., null dereferences, out-of-bounds exceptions). Search-based approach includes EvoSuite [12], a SOTA method that generates regression oracles capable of detecting bugs in future versions of a program. However, this type of oracle cannot detect bugs in the current implementation, as it assumes that an executed behavior is the expected behavior. Specification mining-based methods leverage natural language processing and pattern recognition to derive test oracles from specifications [5, 6, 10, 14, 32, 39]. A recent study has shown that these methods fail to generalize effectively and produce test oracles with subpar bug detection effectiveness [10]. Although the SOTA neural method TOGA was shown to significantly outperform existing neural [40, 41, 43] and specification mining-based methods, and [18] indicates that TOGA generates a high number of false positives and oracles that exhibit poor bug detection effectiveness. TOGLL [17] has shown to outperform TOGA by substantial margin. *Our goal in this paper is to dive deep into investigating the impact of Javadoc comment on the TOG task, rather than focusing strictly on outperforming other TOG methods – though our method did substantially outperform TOGA on real-world fault detection.*

6 CONCLUSION

In this work, we conduct a thorough investigation to understand the impact of Javadoc comments on the test oracle generation (TOG) task. Our large-scale experimental results demonstrate that high-quality Javadoc comments are valuable in automating test oracles, and a minimal prompt consisting solely of Javadoc comments can achieve performance comparable or even better than prompts that include maximum contextual information. Our analysis reveals that the “description” and @return

tags are the most valuable for test oracle generation, and when there are constraints on prompt length, other details can be omitted without significantly impacting accuracy. Furthermore, our study shows that Javadoc can be used to generate oracles that detect faults in real-world programs and, importantly, those oracles outperform oracles generated from MUT implementations. This addresses a significant limitation of prior work, which our methods outperform.

This work is a first step in understanding the potential of Javadoc for TOG. More work is needed to understand how to provide a rich, but minimal, Javadoc context for specific TOG tasks. For example, most Java methods are written in the context of a class and relate to other classes. Exploring methods for identifying the dependencies among code elements, e.g., by following `@InheritedDoc` or `@see` tags, and incorporating Javadoc for all of those elements has the potential to further boost TOG performance.

REFERENCES

- [1] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. 2020. Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 590–601.
- [2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1199–1210.
- [3] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.
- [4] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big?. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*. 610–623.
- [5] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (Amsterdam, Netherlands) (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 242–253. <https://doi.org/10.1145/3213846.3213872>
- [6] Arianna Blasi, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* 181 (2021), 111041.
- [7] Marcel Bohme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 05 (2019), 489–506.
- [8] CodeLlama. 2024. CodeLlama-7B-Instruct Model on Hugging Face. <https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf>
- [9] CodeParrot. [n. d.]. https://github.com/huggingface/transformers/tree/main/examples/research_projects/codeparrot.
- [10] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: a neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141. <https://doi.org/10.1145/3510003.3510141>
- [11] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912.
- [12] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.
- [13] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (dec 2014), 42 pages. <https://doi.org/10.1145/2685612>
- [14] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (Saarbrücken, Germany) (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 213–224. <https://doi.org/10.1145/2931037.2931061>
- [15] Google. 2024. CodeGemma-7B Model on Hugging Face. <https://huggingface.co/google/codegemma-7b>
- [16] Soneya Binta Hossain. 2024. 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 (Lisbon, Portugal) (ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 176–180. <https://doi.org/10.1145/>

3639478.3639791

- [17] Soneya Binta Hossain and Matthew Dwyer. 2024. TOGLL: Correct and Strong Test Oracle Generation with LLMs. *arXiv preprint arXiv:2405.03786* (2024).
- [18] Soneya Binta Hossain, Antonio Filiari, Matthew B. Dwyer, Sebastian Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 120–132. <https://doi.org/10.1145/3611643.3616265>
- [19] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. *Proc. ACM Softw. Eng.* 1, FSE, Article 66 (jul 2024), 23 pages. <https://doi.org/10.1145/3660773>
- [20] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620* (2023).
- [21] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' expectations on automated code comment generation. In *Proceedings of the 44th International Conference on Software Engineering*. 1693–1705.
- [22] Hugging Face. 2024. Hugging Face: The AI community building the future. <https://huggingface.co/>. Accessed: 2024-05-1.
- [23] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440. <https://doi.org/10.1145/2610384.2628055>
- [24] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.
- [25] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.
- [26] Zhongxin Liu, Kui Liu, Xin Xia, and Xiaohu Yang. 2023. Towards more realistic evaluation for neural test oracle generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 589–600.
- [27] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021).
- [28] Paul W McBurney and Collin McMillan. 2015. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2015), 103–119.
- [29] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv:2203.13474* [cs.LG]
- [30] Oracle. [n. d.]. <https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html#format>.
- [31] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
- [32] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*. 815–825. <https://doi.org/10.1109/ICSE.2012.6227137>
- [33] Phi-1. [n. d.]. <https://huggingface.co/microsoft/phi-1>.
- [34] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. 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*. 3505–3506.
- [35] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1105–1116.
- [36] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [37] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).
- [38] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the effectiveness of large language models in generating unit tests. *arXiv preprint arXiv:2305.00418* (2023).

- [39] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 260–269. <https://doi.org/10.1109/ICST.2012.106>
- [40] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
- [41] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. 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*. 54–64. <https://doi.org/10.1145/3524481.3527220>
- [42] Chao Wang, Hao He, Uma Pal, Darko Marinov, and Minghui Zhou. 2023. Suboptimal comments in java projects: From independent comment changes to commenting practices. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–33.
- [43] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409. <https://doi.org/10.1145/3377811.3380429>
- [44] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *arXiv:2202.13169* [cs.PL]
- [45] Michal Zalewski. 2015. American Fuzzy Lop (AFL). <http://lcamtuf.coredump.cx/afl/> Accessed: 29 April, 2024.