

# How Accurately Do Large Language Models Understand Code?

Sabaat Haroon<sup>1</sup>, Ahmad Faraz Khan<sup>1</sup>, Ahmad Humayun<sup>1</sup>, Waris Gill<sup>1</sup>, Abdul Haddi Amjad<sup>1</sup>, Ali R. Butt<sup>1</sup>, Mohammad Taha Khan<sup>2</sup>, Muhammad Ali Gulzar<sup>1</sup>

<sup>1</sup>Virginia Tech , <sup>2</sup>Carnegie Mellon University, USA

{sabaat,ahmadfk,ahmad35,waris,hadiamjad,butta}@vt.edu,tahak@cmu.edu,gulzar@cs.vt.edu

## Abstract

Large Language Models (LLMs) are increasingly being used in post-development tasks such as code repair and testing. A key factor in the successful completion of these tasks is the model's ability to possess a deep understanding of code. However, the extent to which LLMs truly understand code remains largely unevaluated. Quantifying code comprehension is challenging due to its abstract nature and the lack of a standardized metric. Before LLMs, this was typically assessed through developer surveys, which is not feasible for evaluating LLMs. Existing LLM benchmarks focus primarily on code generation, which differs fundamentally from code comprehension. Additionally, fixed benchmarks quickly become obsolete and unreliable as they inevitably become part of the training data.

This paper presents the first large-scale empirical investigation into the ability of LLMs to understand code. Inspired by mutation testing, we use an LLM's ability to find faults as a proxy for its deep understanding of code. This approach is based on the insight that a model capable of identifying subtle functional discrepancies must understand the code well. We first inject faults in real-world programs and ask the LLM to localize them, ensuring the specifications are sufficient for fault localization. Next, we apply semantic-preserving code mutations (SPMs) to the faulty programs and test whether the LLMs still locate the faults, verifying their confidence in code understanding. We evaluate nine popular LLMs on 600,010 debugging tasks automatically sourced across 670 Java and 637 Python programs. We find that LLMs lose the ability to debug the same bug in 78% of the faulty programs when SPM are applied, indicating their shallow understanding of code and reliance on code features irrelevant to semantics. We also find the LLMs understand the code earlier in the program more than the one later. This suggests that LLMs' code comprehension remains tied to lexical and syntactic features due to traditional tokenization designed for natural languages, which overlooks code semantics.

## 1 Introduction

Large language models (LLMs) have traditionally been evaluated for code generation [4, 6]. With the rise of agentic LLMs, their use is rapidly expanding into domains such as debugging [17, 35], code repair [45], and testing [2]. Unlike code generation, post-development tasks like debugging require a deeper semantic understanding of code that is not strictly rooted in lexical and syntactic code features to pinpoint faults. For instance, semantically irrelevant and non-functional code features such as variable names, comments, formatting, documentation (docstrings), and dead code can impede LLMs' ability to reason about the code accurately. Alongside code

generation capabilities, it is equally important to assess LLM's abilities to understand a given code so that they effectively perform follow-up repair, refactoring, and verification tasks.

**Challenges and Problem.** Designing a systematic, scalable method to evaluate LLM's code comprehension abilities presents challenges. First, existing LLM evaluation frameworks for code generation [4, 6, 21, 57] rely on precise specifications encoded as runnable tests, providing clear criteria for measuring effectiveness [4, 6]. In contrast, evaluating an LLM's code comprehension capabilities remains challenging due to its abstract concept and the lack of standardized metrics. Before LLMs, code comprehension research was primarily aimed at improving human developers' understanding, typically assessed through qualitative user studies [12, 37, 38, 65]. Such methods are unlikely to offer the automation and scalability necessary for comprehensive evaluations. Second, even if such manual evaluation was feasible, existing benchmarks such as HumanEval [6] and MBPP [4] do not offer versatility in size and complexity that resemble real-world programs [44]. Third, evaluating an LLM's code understanding requires establishing a reliable ground truth. While existing benchmarks (such as HumanEval [6] and MBPP [4]) have textual specifications against a code snippet that represents the intended code semantics, it is unclear if the specifications are detailed enough to validate the semantics picked by the LLM under investigation. Lastly, even with a robust benchmark, LLMs (e.g., GPT-4 and Claude) undergo continuous upgrades and ultimately train on all benchmarks [36, 52]. Recent efforts to mitigate training data contamination [21, 57] still struggle with LLMs having prior exposure to the code they are expected to generate. To the best of our knowledge, no dataset or method is currently available for code understanding evaluation of LLMs. There is no knowledge of which specific code properties enable LLMs to grasp code semantics better, nor is there an effective strategy to systematically stress-test these models.

**Approach.** This paper presents the first large-scale empirical investigation into LLMs' code comprehension abilities. In the absence of a standardized method and metric for code comprehension, we introduce proxy debugging tasks—localizing faulty lines of code—to assess an LLM's understanding of a given program. Our insight is that if an LLM possesses a fine-grained understanding of a program's semantics against a specification, it should be able to identify deviations from the intended behavior, i.e., a fault. While other post-development tasks, such as repair, specification generation, and testing, also demand code comprehension, debugging is the only task feasible for large-scale automated evaluation. This is because it enables automated validation of an LLM's response against a clear label, i.e., a faulty line of code. In contrast, test generation requires the presence of a comprehensive oracle, and fault repair necessitates an extensive regression test suite.

To this end, our evaluation framework automatically generates diverse and unique debugging tasks from a small set of real-world programs. Since LLMs may perform disproportionately well on programs encountered during training, we ensure the *freshness* of our debugging tasks by dynamically injecting faults and applying semantic-preserving mutations to seed programs, thus mitigating the training data contamination issue. Each debugging task presents a faulty program along with a detailed specification of its expected semantics as input to the LLM, which is then asked to identify the fault. We recognize that an LLM's code understanding is reflected in its sensitivity to (1) faults and (2) semantic-preserving mutations (e.g., comment change, variable names change, and dead code insertion). Fault affects the program's intended behavior, whereas semantic-preserving mutations (SPM) do not.

LLMs are only as effective in debugging as the quality of the specifications they are given. We design a two-phase system. In Phase 1, given a faulty program and its specification, we validate the correctness of the specification through majority voting—excluding faulty programs where *all* LLMs fail to localize the fault. In Phase 2, we introduce an additional layer of specification validation by selecting only those debugging tasks for SPMs that are correctly debugged by the *same* LLMs in Phase 1.

In Phase I, inspired by mutation testing, we systematically inject faults into programs for debugging tasks. Identifying faults represents only one aspect of code understanding. For an LLM to accurately understand a code's semantics, it must also remain insensitive to semantic-preserving mutations that do not affect functionality. Therefore, in Phase 2, we further validate LLM code comprehension by introducing semantic-preserving mutations (SPM) in faulty programs as additional debugging tasks. Our insight behind introducing SPM in faulty programs is as follows: Since LLMs rely on attention mechanisms, similar to Transformers, they unintentionally assign equal significance to non-functional elements, such as comments and dead code. This equal weighting can mislead LLMs, impairing their ability to distinguish semantically relevant code features from irrelevant ones.

**Evaluations.** We operationalize our evaluation framework to assess the code comprehension abilities of 9 state-of-the-art LLMs, including open-source, closed-source, reasoning, coding, and general-purpose models. Using four faults and six semantic-preserving mutations of varying strengths, application locations, and sizes, we automatically generate exactly **600,010** debugging tasks sourced from 637 Python and 670 Java programs. These tasks span **196** million lines of code (LOC), amounting to approximately **3.1** billion tokens being evaluated. We find that, on average, LLMs fail to identify faults in 74% of debugging tasks in cases where at least one LLM correctly detected the same faults. More interestingly, LLMs fail to recognize the same faults they identified in phase 1, in 78% of the debugging tasks, when semantic-preserving mutations (SPMs) are introduced. For instance, function shuffling mutation reduces the debugging accuracy of LLMs by 83% percent. This provides strong evidence that LLMs' ability to understand code semantics is overly influenced by non-functional properties of code. LLMs are most sensitive to inducing dead code SPM and least affected by misleading variable name mutations.

Further analysis reveals that the location of faults and the presence of semantic-preserving mutations play a vital role in LLMs'

```

1 def solveNQueens(n):
2     def is_safe(board, row, col):
3         for i in range(col):
4             if board[row][i] == 1:
5                 return False
6
7         for i, j in zip(range(row, -1, -1),
8                         range(col, -1, -1)):
9             if board[i][j] == 1:
10                return False
11
12     # Off-by-one fault: the loop stops one row too early
13     for i, j in zip(range(row, n-1, 1),
14                     range(col, -1, -1)):
15         if board[i][j] == 1:
16             return False
17     return True
18 <CODE REMOVED FOR BREVITY>
19 def solveQueen(board, col, result):
20 <CODE REMOVED FOR BREVITY>
21 n = 4
22 solveNQueens(n)

```

Figure 1: N-Queen program,  $P_F$ , with an injected fault.

ability to understand code. LLMs tend to have better comprehension of code located earlier in the program. Specifically, 50% of the faults correctly localized by LLMs are in the first 25% of a program's lines, while only 18% are detected in the last 25%.

Overall, across 600K experiments, we find that LLMs still struggle to recognize fine-grained code semantics, limiting their utility in post-development tasks, which are key components of the software development stack [5, 40, 43]. Thus, claims such as "LLMs replacing software engineers" are highly exaggerated. Moreover, we believe a significant effort from the software engineering community is necessary to accurately assess LLMs' capabilities in software engineering tasks, and our study takes a step in this direction.

## 2 Motivating Example

We present a running example based on N-Queen problem taken from the Python code generation benchmark [20]. To illustrate the limitations of LLMs in understanding code, we inject a fault in the program and ask LLM to localize the faulty line of code, with and without semantic preserving mutations, assessing both accuracy and confidence in code understanding. We use GPT-4o in this example. The N-Queen problem involves placing  $N$  queens on an  $N \times N$  chessboard so that no two queens threaten each other; that is, no two queens share the same row, column, or diagonal. This is a fundamental combinatorial optimization problem commonly solved using backtracking and constraint programming. We select the N-Queen problem because its algorithm is well known, and its specification is unambiguous. We verify this by querying LLM about its specifications, which it accurately lists (Figure 2-Left).

For fault injection, we introduce an off-by-one fault in `is_safe` function in line 13 as shown in red text in Figure 1. The fault replaces `n` with `n-1`, which stops the exploration of the *lower diagonal* of the board. The lower diagonal refers to the set of cells that lie diagonally downward (i.e., increasing row indices) and to the left (i.e., decreasing column indices) from a given position. Thus, a queen positioned along this diagonal can attack another queen located at the current position. We call this program  $P_F$ . Next, we

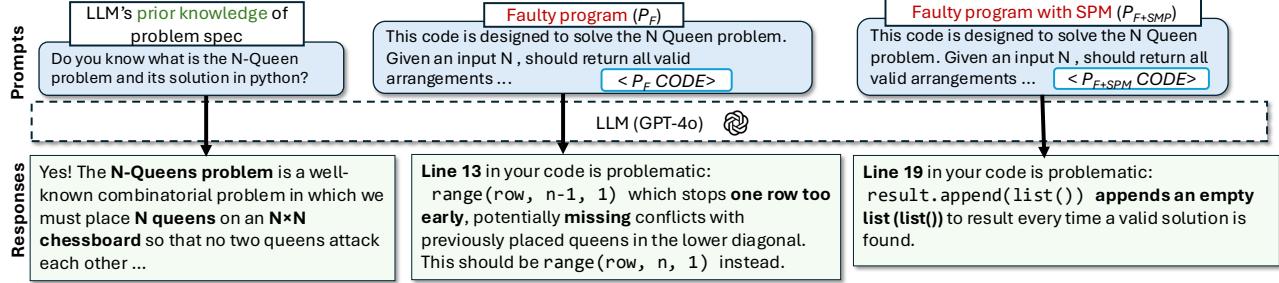


Figure 2: Output of OpenAI’s GPT 4o when asked about the N-Queen problem spec (first), asked for faulty line in the faulty code ( $P_F$ ) (second), and asked for faulty line in the faulty code with semantic-preserving mutations ( $P_{F+SPM}$ ) (third).

create another version of this faulty program by injecting three semantic-preserving mutations: misleading function name, variable name, and comments, shown in Figure 3. Specifically, we rename the function name `solveNQueens(n)` to `howManyQueens(n)` and add a misleading comment (“*This function checks how many queens are on the board.*”) and change variable name `board` to `final_result`. We call this program  $P_{F+SPM}$ . Next, we prepare two debugging tasks for LLM, by first providing  $P_F$  and then  $P_{F+SPM}$ , the original N-queen specification, and ask the LLM to identify the faulty line of code in both. The prompt given to the LLM for both versions of the code remains constant and is shown below:

*“This code is designed to solve the N-Queen problem. Given an input N, it should return all valid arrangements of N queens on an N × N board such that no two queens attack each other. However, the code produces incorrect output. Can you identify the specific line of code responsible for the error? The program is attached below. <CODE>”*

This prompt merges the functional specification with the debugging query to mimic a realistic developer scenario, highlighting the expected behavior and informing LLM about the presence of faults, thereby focusing the LLM on localizing the fault.

For the first debugging task, we ask the LLM to localize the faulty line of code in the faulty program,  $P_F$ . It correctly identifies the line in the code where the fault is located, as shown in the response (Figure2-Middle). To measure how well LLM understood the program, we create the second debugging task by asking LLMs to find the faulty line of code in the program,  $P_{F+SPM}$ . An in-depth, high-quality code comprehension would allow LLM to discard any changes that do not impact the code semantics while continuing to find the fault it identified earlier. However, as shown in Figure 2, LLM cannot identify the faulty line of code in the presence of these semantic-preserving mutations. Instead, it erroneously flags the line 19 with `result.append(list())` as problematic, while the off-by-one error in `is_safe` function remains undetected. *While GPT-4o detects the fault against the program’s specifications, its understanding of the code is easily influenced by semantically irrelevant code and, thus, refraining from identifying the correct impact of changes.*

### 3 Research Questions

To thoroughly assess the code comprehension abilities of LLMs, we formalize our investigation with the following research questions.

```

1 Misleading variable name.
2 def howManyQueens(n):
3     def is_safe(final_result, row, col):
4         Misleading comment.
5         # This function checks how many queens
6         are on the board.
7         for i in range(col):
8             if final_result[row][i] == 1:
9                 return False
10
11        for i, j in zip(range(row, n-1, 1),
12                         range(col, -1, -1)):
13            if final_result[i][j] == 1:
14                return False
15        return True
16 <CODE REMOVED FOR BREVITY>
17     def solveQueen(board, col, result):
18         if col == n:
19             result.append(list())
20             <CODE REMOVED FOR BREVITY>
21 Misleading function name.
22 howManyQueens(n)

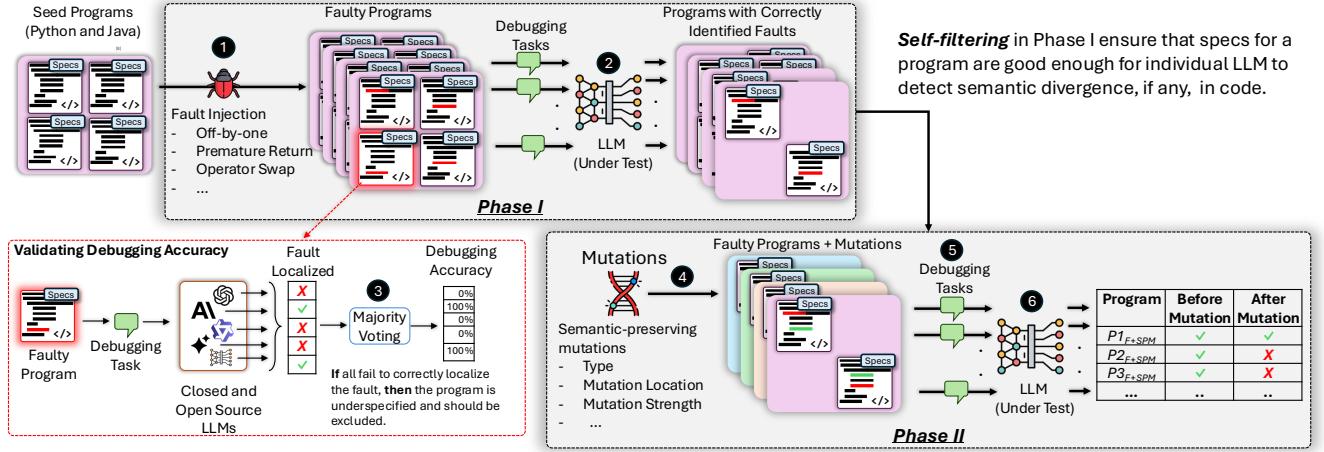
```

Figure 3: N-Queen program,  $P_{F+SPM}$ , with an injected bug in Line 11 and Semantic Preserving Mutations.

- (1) **RQ1: LLMs Code Comprehension:** Do LLMs understand code well enough to identify semantic-altering program faults accurately?
- (2) **RQ2: Confidence of LLMs’ Code reasoning:** Is LLMs’ code comprehension robust and comprehensive enough to be resilient to semantically irrelevant modifications?
- (3) **RQ3: Effect of SPMs’ type and strengths on LLMs’ Code Comprehension:** What specific mutation types and strengths distract the LLMs’ code comprehension?
- (4) **RQ4: Effect of Fault Location on LLM’s Code Comprehension:** How does the location of a bug within a program impact LLMs’ code Comprehension ability?
- (5) **RQ5: Differences Across LLM Categories:** How do different categories of LLMs differ in code comprehension performance under different SPMs?

### 4 Methodology

The goal of this empirical investigation is to generate an arbitrarily large number of debugging tasks dynamically with labeled, correct responses expected from LLMs under test. To that end, we design an automated end-to-end evaluation framework to measure



**Figure 4: Phase I involves injecting faults into seed programs, testing LLMs on debugging tasks, and filtering out underspecified programs. Phase II applies semantic-preserving mutations to assess the confidence with which LLMs understand code.**

an LLM’s code comprehension. Given a set of seed programs, we automatically inject faults in the program in Phase I. We ask LLMs to debug these faulty programs by identifying the injected fault. In Phase II, we take programs from Phase I on which LLM successfully localized faults and automatically inject varying strengths, types, and locations of semantic preserving mutations, creating a large number of unique debugging tasks against which ground truth is available. We assign these debugging tasks in Phase II to LLMs and record their responses while capturing LLMs’ fault localization accuracy in both phases, thus identifying debugging tasks and code characteristics where LLMs have high comprehension.

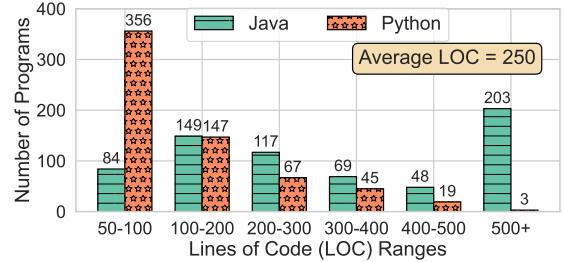
#### 4.1 Seed Programs Procurement

We focus this empirical study on the two dominant languages, Python and Java, used in code generation benchmarks [26]. Both languages are widely used in open-source projects [33], consequently providing adequate training opportunities for LLMs.

We use the following criteria to identify the seed programs. First, the programs must accompany a natural language description of the expected specification and semantics of the program. This is a strict criterion for preparing well-defined debugging tasks since, without specifying the expected semantics of the code, the notion of correctness and incorrectness will remain unclear. Second, the program must have at least 50 lines of code. Recent studies show that LLM benchmarks contain small (< 50 LOC) toy programs that do not represent real-world programs [11]. Since most of these benchmark programs are publicly available, the LLMs are also highly likely to have prior knowledge of these programs, leading to disproportionately high accuracy in code generation. Third, we must not use already faulty programs or part of a faulty benchmark (e.g., Defects4J [25] or buginPy [59]) as LLMs might have prior knowledge of those faults. Lastly, the complete size of each program should not exceed the prompt size limits of the current LLMs APIs.

We find two public benchmarks of Python and Java that satisfy the criteria above. For Python, we obtain the programs from [20]. For the Java programs, we gather programs from CodeSearchNet [51]. The dataset consists of 18612 Python programs and 812

Java programs. After applying the size and context limit filter, we get the final set of 637 Python and 670 Java programs. Figure 5 presents the line-of-code (LOC) distribution of these programs, with an average LOC of approximately 250 across both languages.



**Figure 5: LOC Distribution for the Final Set of Programs**

#### 4.2 Phase I: LLMs’ Code Comprehension

In Phase I, we perform specification validations from two aspects: intra-LLM-based and inter-LLM-based validations. Intra-LLM validations ensure that the LLM understands the program with respect to the specification before testing the robustness of its code comprehension (Phase II). Inter-LLM validation checks if an incorrect fault localization is due to low-quality specifications (i.e., most LLMs do not localize fault) or an LLM’s inability to understand code (i.e., most LLMs localize fault). The first step in Phase I is injecting a fault (1 in Figure 4) and asking an LLM to identify the fault (2 in Figure 4) by providing the faulty program and its specifications. If the LLM correctly identifies the fault location, it verifies that the injected fault is indeed incorrect as per the specification, showing its understanding of the code, and such faulty programs go to Phase II.

If the LLM does not correctly localize the injected fault, it could be either due to (1) the program being underspecified or (2) the LLM’s inability to understand the code and match its semantics with the specifications. To distinguish between the two cases, we perform the second step in Phase I, shown in 3 in Figure 4. We

Modification	Type	Description	Example
Off-By-One	Fault-Inducing	Alters the loop range, causing an off-by-one error.	for i in range(n) → for i in range(n+1) Inserting return before processing all elements. a && b → a    b a + b → a - b if(False): x = 5
Misplaced Return	Fault-Inducing	Adds a return statement at an unintended location, leading to early termination.	/* Summon ancient dragons */ count → index
Boolean Logic Operator Swap	Fault-Inducing	Switches boolean operators	
Dead Code Injection ( $\mathcal{M}_d$ )	Semantic-Preserving	Switches arithmetic operators	
Misleading Comments ( $\mathcal{M}_c$ )	Semantic-Preserving	Adds code that does not execute or is unused. This increases complexity without changing semantics.	
Misleading Variable Names ( $\mathcal{M}_v$ )	Semantic-Preserving	A comment is replaced with misleading but linguistically coherent descriptions.	
Function Shuffling ( $\mathcal{M}_f$ )	Semantic-Preserving	Variable names are replaced with ones that obscure their real function.	void fA(); void fB(); → void fB(); void fA();
		The order of function definitions is changed without affecting their dependencies. Since our Python dataset only contains single function code this mutation is only developed for Java.	

**Table 1: Types of Faults and Mutations Applied to Seed Programs**

Model	Size
<b>Open-Source Models</b>	
Qwen2.5-coder [18]	7B
Phi4 [1]	14B
Llama3.1 [16]	8B
Qwen-QWQ [54]	32B
Deepseek v3 [30]	671B
<b>Closed-Source Models</b>	
GPT-4o [19]	Undisclosed
Claude 3.7 Sonnet [3]	Undisclosed
Gemini 2.0-Flash [10]	Undisclosed
Gemini 1.5-Pro [9]	Undisclosed

**Table 2: LLMs Evaluated**

perform differential analysis for programs where an LLM could not localize the injected fault by sending its debugging task to 9 state-of-the-art LLMs and performing majority voting. If all LLMs could not locate the fault, then it is likely that the program is underspecified. If one or more of these LLMs correctly localize the fault, the specifications are adequate to assist other LLMs in extracting faults. This Inter-LLM-based validation helps determine the code understanding of the LLMs under test and eliminate cases where programs are underspecified.

*Fault injection process.* We adopt fault from standard mutation testing analysis research [24]. We focused on (1) faults that are confined to a single line, (2) faults pertaining solely to the logical elements of the top-level code and cannot stem from dependencies, and (3) faults that should affect the program’s logic rather than introduce syntax errors. We consciously select a small number of *simple* faults to ensure consistency across experiments and fairness for LLMs. Since LLMs today process code sequentially, token-by-token, *their understanding of code may vary across different program locations*. To discover further evidence, we randomly selected program locations from the 0-25%, 25%-50%, 50%-75%, and 75%-100% lines of code to inject fault. Table 1 presents the types of injected listed under the Type “fault-inducing” mutations. For instance, incorrect boolean logic and arithmetic operators swap are LOR and AOR mutation operators from MAJOR [24], whereas premature return and off-by-one are from [46, 48].

*Fault detection process:* After fault injection, we provide a set of nine different LLM models with a prompt containing the program, its natural language specification, and the task. Table 2 provides details of the LLMs we used. Once the LLM provides us with the line at fault, we automatically compare it with the recorded line number during fault injection. Faulty programs where the LLM correctly identified the fault are used for Phase II.

### 4.3 Phase II: LLM’s Code Comprehension Confidence

Phase II evaluates the robustness of the LLM’s understanding of the code by applying semantic-preserving mutations to faulty programs where it previously localized the faults correctly. Our insight is that if the LLM adequately understands the code, it should be able to ignore the semantically irrelevant changes that do not influence program functionality and thus maintain its debugging

performance. The benefits of this process are twofold. First, these semantic-preserving mutations provide us with a step-wise dial to evaluate the limits of the LLM. Second, semantic-preserving mutations reaffirm that the LLM under test is not just syntactically comparing the program (i.e., a form of shallow reasoning) with a version it has seen in its training data.

On programs that an LLM has successfully localized fault before, we apply semantic preserving mutations of different characteristics (❷ in Figure 4), prepare a debugging task (❸ in Figure 4), and ask LLM to localize the same fault again without any context of the previous debugging task (❹ in Figure 4). Correctly localizing the same fault shows the LLM’s deeper and more robust understanding of code compared to cases when it does not localize the fault, indicating shallow code reasoning. We develop four categories of semantic-preserving mutations:

- **Annotative:** Changes to non-executing parts of the code, such as comments, annotations, or metadata. They help evaluate how much LLMs rely on annotations or documentation for analysis.
- **Identifier:** Changes to the names of variables, functions, or other identifiers. This tests the resilience of LLMs and whether their reasoning is tied to concrete code structure rather than abstract.
- **Structural:** Changes to the program structure that do not modify functionality. For example, inserting unreachable statements. This helps assess if LLMs ignore semantically irrelevant code.
- **Non-additive:** Changes to the code order without introducing new content or changing semantics. Examples include changing the order of function declarations.

For these four categories, we develop one representative mutation each, summarized in Table 1 under Type “Semantic-Preserving”. For Dead Code Injection, Misleading Comments, and Misleading Variable Names, the core elements, e.g., the content of the comments, the names of the variables, and the snippets of dead code, are generated via the LLM under test. We insert these components via AST manipulation. During the insertions, we continuously track the movement of the original fault, allowing us to still reliably identify faulty lines, even if their position shifts due to formatting changes.

*Mutation Applications.* We generate multiple program variants by applying mutations both individually and in combination. This means given a program  $\mathcal{P}$ , we construct a total of six program mutants. Using the notation in Table 1, four mutants are obtained by applying the individual mutations:  $\mathcal{M}_c(\mathcal{P})$ ,  $\mathcal{M}_v(\mathcal{P})$ ,  $\mathcal{M}_d(\mathcal{P})$ , and  $\mathcal{M}_f(\mathcal{P})$ ; and two more are obtained by composing:  $\mathcal{M}_c(\mathcal{M}_v(\mathcal{P}))$  and  $\mathcal{M}_c(\mathcal{M}_v(\mathcal{M}_d(\mathcal{P})))$ .

Fault Type	Gemin 2.0	Gemin 1.5	GPT 4o	Llama 3.1	Phi-4	Owen 2.5- coder	Owen QwQ	Claude 3.7- Sonnet	Deepseek v3
Python Benchmark Programs									
IncorrectBooleanLogic	5.54	8.92	6.87	3.25	5.42	7.78	3.61	9.88	<b>25.96</b>
MisplacedReturn	43.28	55.11	74.62	17.33	44.60	16.11	20.74	<b>83.33</b>	29.31
OffByOne	15.60	15.02	14.39	7.80	9.29	<b>16.75</b>	8.37	16.00	11.28
OperatorSwap	66.00	<b>87.78</b>	68.22	37.11	52.22	14.02	29.11	85.78	38.16
Java Benchmark Programs									
IncorrectBooleanLogic	16.26	23.54	13.59	6.07	9.95	10.44	9.22	<b>48.79</b>	13.04
MisplacedReturn	3.33	39.63	36.37	6.87	15.77	15.77	8.50	<b>70.83</b>	9.56
OffByOne	16.67	33.33	37.50	8.33	8.33	4.17	4.17	42.22	20.00
OperatorSwap	29.65	30.52	15.41	11.48	14.10	20.64	4.51	<b>73.26</b>	8.73

**Table 3: Detection accuracy of different LLMs on various bug types in Python and Java in Phase I.**

For each mutation application, we randomly select program location from 0-25%, 25%-50%, 50%-75%, and 75%-100% percentile of the lines of the code to apply the mutations. We also change the strength of the mutation, e.g., the number of lines of dead code. By layering mutations incrementally, we create a structured progression of debugging difficulty. This structured approach ensures a gradient of debugging difficulty, ranging from minor annotation inconsistencies to major logical confusion.

*How LLMs are prompted.* The resulting debugging tasks are sent to nine different LLMs shown in Table 2, selected based on their popularity. The choice of nine models ensures a diverse evaluation, incorporating widely used proprietary and open-source alternatives. Open-source models were executed on local server machines with high-end GPUs, leveraging local computational resources to allow greater control over inference parameters and reproducibility. Meanwhile, closed-source models were accessed via their respective APIs. Once a model returns the predicted line number, it is compared to the expected faulty line position after mutation.

## 5 Results

We analyze the performance of the LLMs across various dimensions, including fault type, fault location, language, SPM type, SPM strength, SPM location, and LLM category. We create a total of 600K debugging tasks for these LLMs, spanning 196 million lines of code, and 3.1 billion tokens. Table 4 documents the experiment statistics.

Metric	Value
Total Prompts Generated	930,949
Total Debugging Tasks	600,010
Average LOC Tested	250 LOC
Total LOC Tested	196,685,313 LOC
Total Token Analyzed	3,105,262,000
System Specification APIs	48 GB RAM, 48 cores with NVIDIA L40s GPU Gemini, Anthropic, Deepseek, OpenAI

**Table 4: Experimental Statistics**

### 5.1 RQ1: LLMs' Code Comprehension

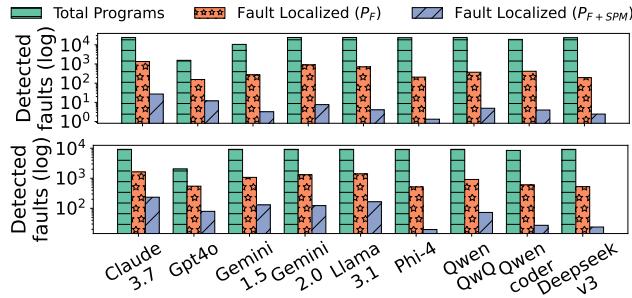
We analyze the accuracy of individual LLMs in finding the faulty line of code. As mentioned below, verifying if the specifications are adequate enough to distinguish the impact of the injection is highly challenging and nearly infeasible at the scale of our experiments. To automate this process, we perform *majority voting* to ensure the quality of the specification and exclude faulty programs to measure fault localization where all LLM did not correctly localize the faults. Table 3 presents these results. We measure accuracy in

these experiments as the ratio of successful debugging tasks to the total number of faulty programs. For testing with the Deepseek v3 model, we utilized a portion of our originally created dataset. In particular, we use 5% of the debugging tasks from the entire dataset for Deepseek due to the relatively slow average response time of 23 seconds per API call. Given this, analyzing the entire dataset would be time-prohibitive. Additionally, since LLMs' accuracy is low in Phase I, we applied a majority voting approach, including only those debugging tasks detected by more than four LLMs in the 5% sample. Overall, fault detection accuracy in Java programs is significantly lower than in Python programs with the exception of Claude. For example, for fault type *MisplacedReturn*, we find GPT-4o accurately localizing fault in 74.62% of faulty Python programs, whereas this accuracy drops down to only 36.37% for Java programs. Java is statically typed with relatively more semantic information encoded in the syntactic structure of the program than Python, which is dynamically typed. Therefore, LLMs have more opportunities to extract partial semantics from the code. However, the results of all other LLMs except Claude contradict this. Since Claude is the only model to contradict this, it is likely caused by a difference in training datasets for Claude where Claude performs well overall on Java compared to Python. Claude is the best-performing LLM for Java in Phase I.

For *Incorrect Boolean Logic*, most LLMs provide higher fault localization accuracy on Java programs. The same behavior is seen with closed-source LLMs (i.e., Gemini, Claude, and GPT) and open-sourced LLMs like Deepseek v3 for fault *OffByOne*. A common property between the two is that they will likely influence the branching decision. Since Python uses indents to separate code blocks compared to braces in Java, LLMs struggle with the code branching logic in Python. Java explicitly expresses code branching with tokens like "{" and "}" that LLM successfully picks and thus captures the branching logic better. Therefore, faults that directly impact the code branching in Java are localized with higher accuracy by most LLMs. In *OffByOne* faults, open-source LLMs have lower accuracy when the fault was injected in Java compared to Python, which can be attributed to the generally low reasoning capabilities of LLMs with verbose and imperative languages like Java as compared to the declarative nature of Python.

Across different fault types, the highest detection accuracy achieved by any LLM ranges from 3.25% to 87.78% for Python programs and from 3.33% to 73.26% for Java programs, highlighting a gap of more than 84% in the best detection accuracies. Among all fault types, *OperatorSwap* and *MisplacedReturn* are the most easily detected, with accuracies of 87.78% and 83.33% for Python and 73.26% and 70.83% for Java respectively. This is expected because both faults introduced textually noticeable changes in a program path that are most likely to cause an explicit behavioral change (e.g., terminating a method earlier). In contrast, the other two faults introduce changes that implicitly affect the branching logic (e.g., not executing the last iteration of a loop).

*LLM Performance Comparison: Open-Source vs. Closed-Source.* We identify the top-performing models in each category. Notably, Claude 3.7 Sonnet and Gemini 1.5 Pro lead in fault detection performance, demonstrating high code comprehension, followed closely by OpenAI's GPT-4o. The highest fault detection accuracies come



**Figure 6: Effect of mutations on fault detection accuracy across all fault types on each LLM with the Java dataset (top) and Python dataset (bottom).**

from closed-source models, suggesting that proprietary LLMs benefit from extensive training data, deployments on hardware needed to host the full-scale models, and optimizations. Among open-source LLMs, Qwen2.5-coder:7B, Deepseek v3, and Phi-4:14B perform best, with Qwen2.5-coder achieving the highest fault detection accuracy (16.75%) for the OffByOne fault and Deepseek v3 achieving the highest fault detection accuracy (25.96%) for IncorrectBooleanLogic in the Python dataset. Deepseek v3, Qwen2.5-coder:7B, and Phi-4:14B outperform other open-source models. Although these models are not specialized for code-specific tasks, they benefit from being trained on a larger and more recent corpus of data compared to older generic models like Llama 3.1. This fresh training data grants them a richer understanding of modern coding practices and language nuances. In contrast, Qwen-QwQ, despite its impressive 32B parameter size and strong reasoning capabilities, appears to overthink code-related tasks and misanalyze the fault [47].

## 5.2 RQ2: Confidence of LLMs’ Code Reasoning

We perform a resiliency assessment on the code comprehension ability of LLMs by mutating faulty programs with semantic-preserving changes. This experiment aims to measure the depth to which LLM understood the speciation and code and how confident it is in its understanding of the code.

**5.2.1 Fault Localization After Applying Mutations.** First, we examine the impact of semantic-preserving mutations (SPMs) on fault localization accuracy by asking LLM to localize the same fault in a faulty program ( $P_F$ ) and in a mutated faulty program ( $P_{F+SPM}$ ). We use six types of SPMs (from Table 1). We also apply cumulative mutations to increase the difficulty of debugging tasks: Variable Cumulative (which combines both misleading comments and misleading variable mutations) and Dead Code Cumulative (which incorporates all three mutation types).

Figure 6 summarizes the results, averaged across all fault types and SPMs. We measure accuracy in these experiments as the ratio of successful debugging tasks in Phase II to those in Phase I. Almost every model showed a notable drop in fault localization accuracy when SPMs were present. Open-source models such as Llama3.1:8B and Qwen models experienced the largest declines, whereas closed-source models like Claude, GPT-4o, and Gemini

demonstrated greater resilience. This performance gap is largely due to the models’ reliance on surface-level cues such as variable names, comments, and code structure, which they use to infer underlying semantics. When these cues are manipulated without altering the functionality, the models’ learned representations become less reliable. In contrast, closed-source models benefit from extensive fine-tuning on high-quality, diverse datasets, which helps them build deeper semantic understanding and robustness against such perturbations.

Overall, the difference in fault detection accuracy between original and mutated programs highlights that all LLMs are highly sensitive to semantic-preserving mutations that do not alter functionality.

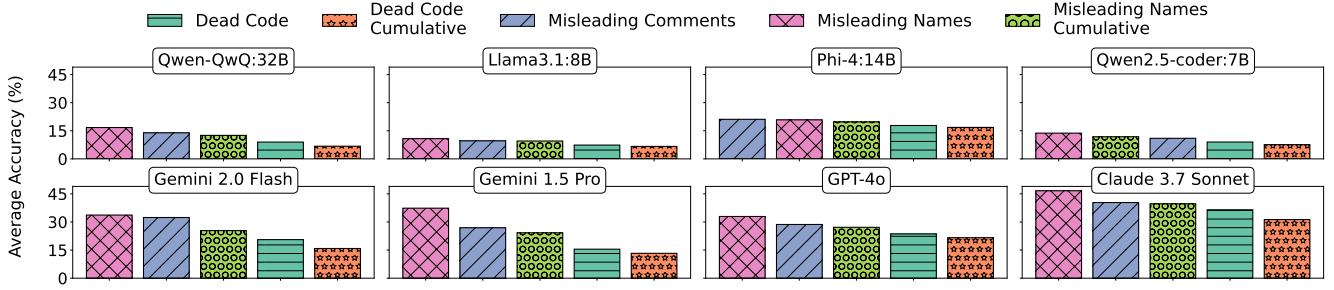
**5.2.2 Language-specific Performance under Mutations.** This subsection addresses how the effect of mutations varies by programming language. Figure 6 shows the performance breakup by language with results of Java (top) and Python (bottom), revealing discrepancies for different languages when detecting faults with SPM. The results show that it is considerably challenging to detect faults in Java compared to Python across all models; the difference is significantly visible, especially for the Claude, GPT-4o, and Gemini models. Deepseek v3’s shows a similar trend that detecting faults in Java is more challenging in comparison to Python and the fault detection accuracy drops further in presence of SPMs.

We attribute this performance difference to the size and diversity of the training data available for these models, as well as the intrinsic structural differences between Java and Python. Prior work evaluating large language models trained on code has highlighted that richer and more diverse training datasets can significantly enhance model performance in code-related tasks [7]. Python’s concise syntax and its widespread use in scripting and data science lead to more comprehensive coverage in these datasets, providing models with richer contextual knowledge [50]. Conversely, Java’s verbose and strictly object-oriented nature can pose additional challenges for fault detection, especially for higher-level logical errors overshadowed by SPM. Furthermore, differences in coding conventions and library standardization contribute to more uniform fault patterns in Python, aiding the detection process [56]. Consequently, while all models exhibit reduced performance for Java compared to Python, the gap is most pronounced for GPT-4o and Gemini, unlike Claude, which retains its performance. This suggests that unlike Claude, GPT-4o and Gemini may be more sensitive to variations in language-specific code structures.

## 5.3 RQ3: Effect of Mutation Characteristics

We further break down the results from Phase II code comprehension assessment into mutation type and mutation strength and assess how each affects the fault localization of LLMs, assessing LLMs’ confidence in code comprehension.

**5.3.1 Effect of Mutation Type on LLM’s Code Comprehension.** Figure 7 presents the average fault localization accuracy across different LLMs for different types of SPMs in Phase II. Across all tested LLMs, *Misleading Variable Names* causes the least impact (28.7% accuracy) on the fault localization accuracy, suggesting that models often rely on contextual cues beyond variable names. One reason



**Figure 7: Effect of mutation type on fault detection accuracy averaged for Python [20] and Java datasets [51].**

is that the variable renaming maintains the structural and lexical integrity of code and that the training data for LLMs have variable names of all types as there is no single variable naming convention, and therefore, LLMs are relatively more robust against those.

Interestingly, *Misleading Comments* result in lower average Phase II accuracy (24.55%) compared to *Misleading Variable Names* (28.7%), indicating that LLMs often try to extract a code’s semantic information from the comments in addition to the code itself, even though comments have no effect on code execution and are removed during compilation. For example, a program from the Python seed programs implements a grid-based game using Pygame. The code contained a misleading comment stating, “Create grid: This grid is auto-populated with unique, random values for each cell, ensuring a dynamic game board.” This comment implied a specific behavior in grid initialization. The fault was an off-by-one error in the loop (using `for row in range(rows - 1):` to traverse the grid), which caused the grid to be traversed one cell fewer than intended. Notably, Gemini 1.5 Pro inaccurately predicted the fault in this case, highlighting its vulnerability to being misdirected by incongruent information provided in code comments. Additionally, *Dead Code* has a more substantial impact on accuracy by introducing extra code that changes the program’s structure without affecting its functionality. LLMs’ fault localization accuracy drops to 18.5%. For example, a program from seed programs simulates an autonomous car, the update method contains the injected fault (Phase I) where the car’s vertical movement is adjusted with an incorrect offset (using `self.rect.y += self.change_y - 1`). The program contains an unreachable dead code block that defines a function for debugging sensor values. This dead code changes the syntactic structure without affecting functionality, distracting the model (Phi-4) and leading to an inaccurate prediction of the line with the fault. Similar observations are made when evaluating the DeepSeek V3 model, as shown in Figure 8, using sampled dataset (same as in Figure ??, comprising 5% of the debugging tasks). This is expected, as LLMs have a shallow understanding of code,

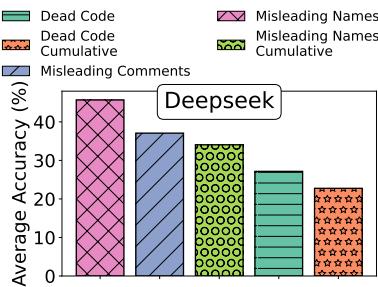
which is insufficient for performing precise code reachability

analysis and identifying unreachable code that is irrelevant to the program’s functionality. Instead, their limited code comprehension leads them to treat such code as actively contributing to the program’s semantics, thereby influencing fault localization. Compounding multiple mutations significantly degrades LLMs’ fault detection Phase II accuracy, with *Variable Cumulative Mutation* (22.67% accuracy) and *Dead Code Cumulative* (15.77% accuracy).

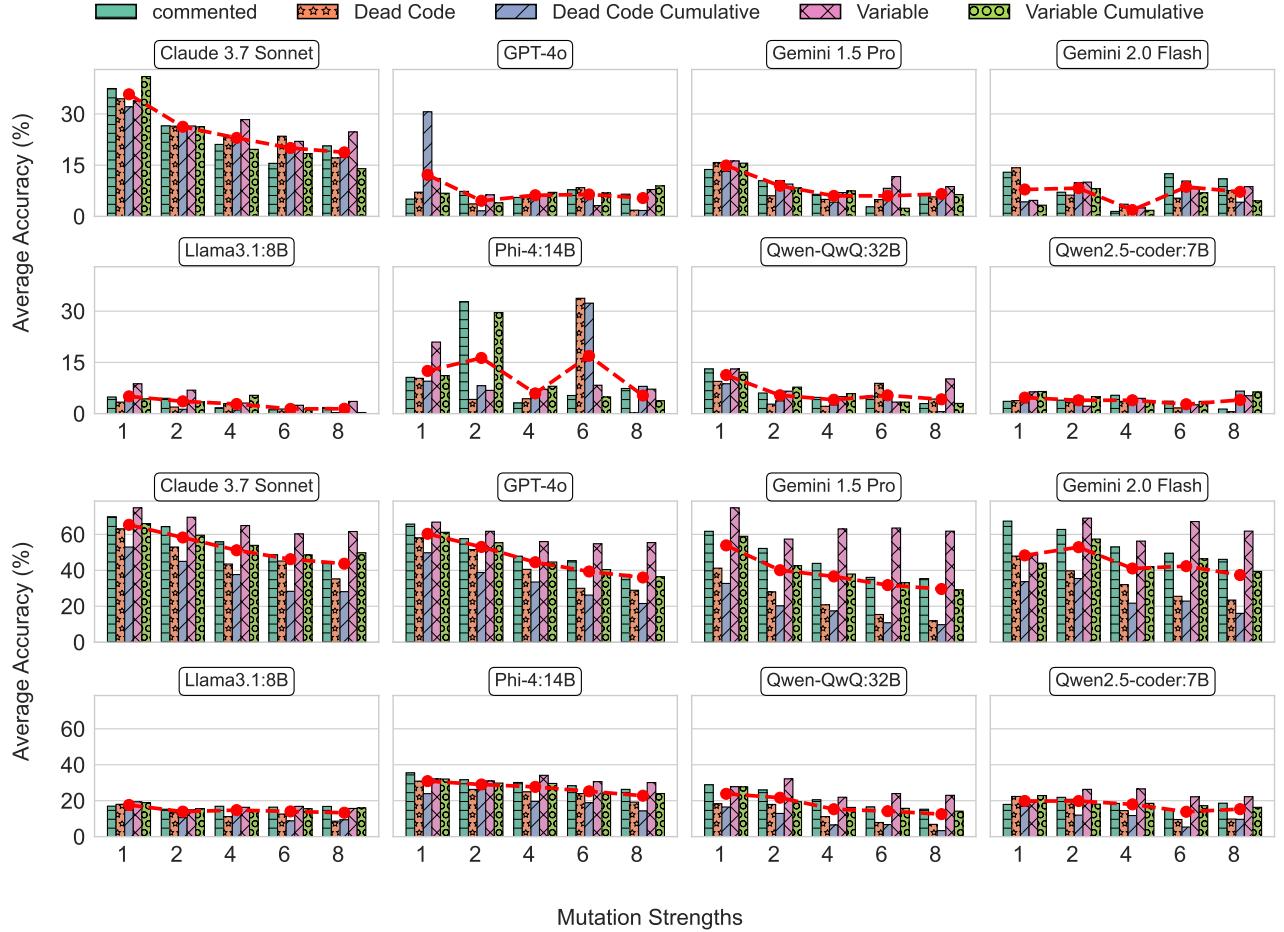
**5.3.2 Effect of Function Shuffling on Bug Detection Accuracy.** Function shuffling mutations only apply to Java programs. Across four fault types, we observe an accuracy drop of 83%, demonstrating that even without introducing any new code, the ability of LLMs to understand code is significantly compromised. This coincides with the results for RQ4 that the location of the code plays a prominent part in the code comprehension ability of LLMs.

**5.3.3 Effect of Mutation Strength on LLM’s Code Comprehension.** Figure 9 presents the results for each mutation type with varying strength levels (1 to 8), where strength represents the number of times an SPM was applied within a single program. For most LLMs, we find a linear decline in average fault localization accuracy (highlighted by the red line) as the strength of SPM increases. This trend occurs because, as more SPMs are injected, the accumulated syntactic noise progressively obscures the underlying code semantics, making it harder for the models to extract meaningful contextual cues. At lower strength levels, LLMs can often rely on reliable hints from variable names, comments, and code structure to localize faults correctly.

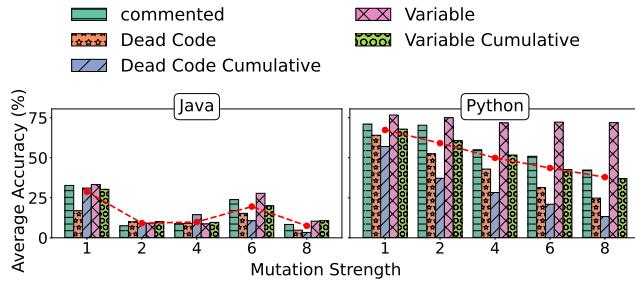
For instance, in Java, the accuracy drops on average from 14.79% at strength 1 to 6.71% at strength 8 across all evaluated models, corresponding to an overall decrease of 8.08% over 7 steps (an average decrease of 1.15% per mutation strength increase). Similarly, in Python, the accuracy falls on average from 43.03% at strength 1 to 27.6% at strength 8 across all evaluated models, which represents an overall decline of 15.43% over 7 steps (an average decrease of 2.2% per mutation strength increase). For example, one of the seed Python programs defines a function called `make_tile` using the Blender Python API to create a 3D model of a tile. In this program, a fault is injected by doing `return None` early, preventing the execution of the subsequent parts of the function. At a lower SPM strength with a single dead code line, the LLM (Gemini 1.5 Pro)



**Figure 8: Effect of mutation type on fault detection accuracy averaged for sampled Python [20] and Java datasets [51].**



**Figure 9: Effect of mutation strength on fault detection accuracy across each mutation type for different models with the Java programs (top two rows) and Python programs (bottom two rows). The red line gives the trend of average accuracy for all mutation types across mutation strengths. Accuracy is the ratio of successful fault localizations in Phase II to those in Phase I.**



**Figure 10: Effect of mutation strength on fault detection accuracy across each mutation type for Deepseek v3 models with the filtered Java and Python programs. The red line gives the trend of average accuracy for all mutation types across mutation strengths. Accuracy is the ratio of successful fault localizations in Phase II to those in Phase I.**

locates the fault. However, when the strength of the dead code SPM was increased to 4 by introducing multiple dead code statements, it ultimately obscured the injected fault. As a result, the LLM could

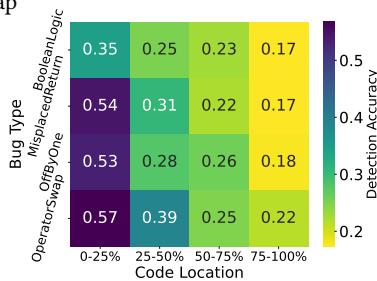
no longer accurately identify the line with the fault. Similar results can also be observed for the Deepseek v3 model using the same dataset as in Figure 10. This Figure shows a downward trend in average accuracy as mutation strength increases especially for the Python dataset, revealing the compound effect of SPM intensity on fault detection accuracies. This is a prime example of LLMs lacking the depth in semantic understanding required to find faults in the presence of SPMs.

#### 5.4 RQ4: Effect of Fault Location in the Code

Since LLMs process code a sequence of tokens, extracting semantics from primarily textual information, we hypothesize that LLMs may have varying levels of code understanding at different program locations. Prior research on LLMs' reasoning and context retention shows strong evidence that they lose context as the sequence length or context window expands [53]. In code reasoning, this could translate into lower code comprehension for certain code regions. We design the following experiment to validate this hypothesis. We divide each program into four equal sections

based on length (0 – 25%, 25 – 50%, 50 – 75%, and 75 – 100%) and analyze LLM’s fault localizability with a fault in each segment.

Figure 11 presents a heatmap summarizing fault localization accuracy across different fault types and their positions within the code. Results show that faults in the first quarter of the code (0 – 25%) are detected with the highest accuracy, suggesting that LLMs may focus more on initial segments and retain clearer context at the beginning of a program. In particular, *OperatorSwap* faults are easily detected in this early section. However, detection accuracy declines as faults appear later in the code, with the 75 – 100% range exhibiting the lowest success rates across all fault types. This decline is most pronounced for Misplaced Return faults, indicating that this error type is not only difficult to detect overall but become even more challenging when positioned later in the program. These findings suggest that LLMs may lose contextual cues or allocate less attention to code segments appearing farther from the start.



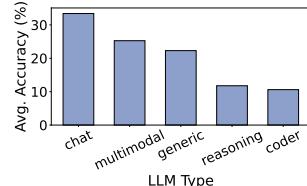
**Figure 11: Effect of fault location on fault detection accuracy.**

## 5.5 RQ5: Categories of LLMs

Figure 12 provides an aggregated view of LLM performance in fault detection, showing that multimodal models and Mixture of Expert Chat models (e.g., Gemini 1.5 Pro, GPT-4o, and Deepseek v3) achieve the highest accuracy, while coding-specialized and reasoning models perform the worst.

This result suggests that coding-focused LLMs rely heavily on past knowledge of training programs, which becomes less useful when encountering dynamically mutated code in our benchmark. The performance loss in reasoning models can be attributed to overthinking. For instance, in a few Python programs within our dataset, Qwen-QWQ predicted faulty line numbers that exceeded the total lines of code, clearly overanalyzing the problem and causing its chain of thought to derail and produce infeasible solutions. However, this issue was not observed in any other LLM.

These findings align with studies on reasoning models in code assistance, showing that overthinking can lead to analysis paralysis and rogue actions [8, 47]. While reasoning models perform worst in fault detection, structural reasoning could improve accuracy [29], but we exclude such approaches due to manual intervention. In contrast, generic and multimodal models interpret the altered program’s specifications better, likely due to their broader training data and better understanding of natural language tasks.



**Figure 12: Effect of LLM Type on fault detection accuracy.**

## 6 Discussions

The empirical investigation reveals fundamental weaknesses in state-of-the-art LLMs, specifically their sensitivity to non-functional code changes. Introducing semantic-preserving mutations (SPMs) shows that even subtle, non-behavioral alterations significantly reduce debugging accuracy. This sensitivity poses practical reliability concerns, as developers frequently refactor code, adjust formatting, update documentation, or leave dead code. We intentionally avoid extensive prompt engineering, using a uniform baseline prompt and altering only the faulty program (illustrated in Figure 2). Although techniques like interactive multi-shot prompting and iterative refinement might enhance performance, they require significant manual intervention and are unsuitable for large-scale evaluations. Additionally, our findings highlight security implications, particularly regarding prompt injection attacks. The sensitivity of LLMs to semantic-preserving code changes indicates potential vulnerabilities to subtle manipulations. Recognizing these susceptibilities helps researchers and practitioners proactively identify attack vectors and strengthen model resilience. Currently, LLMs process code similarly to textual data, relying on generic tokenization methods that overlook code-specific structures. Our results have exposed this gap in LLMs’ processing of code similar to natural languages. We argue that converting code into intermediate, structured representations using language-specific lexical and syntactic parsing could significantly enhance LLM reasoning capabilities. This approach can better capture syntactic and semantic nuances, improving robustness to non-functional code alterations and generalization across comprehension tasks. Future research should explore integrating representations such as Control Flow Graphs (CFGs) and Code Property Graph (CPG) [63] into LLM frameworks for improved reasoning performance. LLMs working at a unified abstract layer like CPG can also open opportunities for cross-language reasoning.

**Threats to Validity.** While our method rigorously evaluates LLMs’ code comprehension through extensive debugging tasks, it is not exhaustive. We do not claim that LLM fully understands every aspect of the target code when it localizes a fault. Future research can integrate program analysis techniques like program slicing to introduce faults along different execution paths, ensuring high path coverage. While we use debugging tasks as a proxy for assessing code comprehension, they are not the only means of evaluating an LLM’s understanding. Alternative approaches, such as specification or invariant inference, elements of which have been explored in other works [34, 42, 61, 62]. However, these methods often face challenges in obtaining ground truth, either due to procurement bottlenecks or difficulties in automated evaluation. Given these constraints, debugging remains the most scalable approach. Since we evaluate nine LLMs, the results of this empirical study may not generalize to all current and future LLMs. However, to mitigate this threat, we cover a diverse range of both open- and closed-source models, including both general-purpose and coding-specific LLMs. Similarly, our debugging tasks are sourced from Java and Python programs. Therefore, the presented findings may not apply to other languages or programming paradigms. We selected Python and Java primarily due to their widespread usage and the strong performance of LLMs on these languages.

Benchmark	Ground Truth	Multi-Lang	CCC	Scalability	RDC
HumanEval+ [31]	✓	✗	✗	✗	✗
DebugBench [55]	✓	✓	✗	✗	✗
BugsInPy[59]	✓	✗	✗	✗	✗
LiveBench [58]	✓	✓	✗	✗	✓
Ours	✓	✓	✓	✓	✓

**Table 5: Comparison of benchmarks under five criteria:** (1) *Ground Truth*, (2) *Multi-Lang*, (3) *Configurable Code Complexity (CCC)*, (4) *Scalability*, (5) *RDC* (robust to data contamination). Ticks (✓) or crosses (✗) are inferred from references.

## 7 Related Work

Evaluating large language models (LLMs) on code-related tasks is an active area of research, spanning from code generation and synthesis to debugging and fault localization. We organize the related literature into four main categories.

*Code Generation, Comprehension, and LLM Performance.* Earlier studies have focused on the capabilities of LLMs in synthesizing code and ensuring its syntactic correctness. Daye et al. [38] propose a plugin powered by LLMs to help users understand code snippets. One of the evaluation criteria for this plugin is assessing the LLM’s code understanding before it assists developers in comprehension. However, their approach primarily relies on humans verifying the LLM-generated explanations against the actual code, which prevents large-scale code comprehension evaluations. Le et al. [27] investigates the interplay between natural language understanding and code synthesis. Fakhoury et al. [12] evaluate code generation alignment with user intent with a user study of 15 programmers. Liu et al. [32] proposes EvalPlus to benchmark the functional correctness of LLM-synthesized code. Across these efforts, the primary challenges of having an automated evaluation methodology remain unaddressed. In terms of inferring program specification, recent work [42, 61] evaluate LLM performance for inferring various program invariants while others [13, 34, 62] explore generating formal program specifications using LLMs. Although these studies have advanced our understanding of LLM performance in generating functional code, they either neglect the deeper issue of whether such models truly capture the semantics, or they fail to conduct the study at scale due to dataset limitations.

*Debugging and Fault Localization.* Recent research [28, 39, 49] utilize and assess debugging capabilities of LLMs. They leverage automated fault localization strategies using mutation testing and functional test cases to determine whether an LLM can pinpoint errors. However, common limitations in their evaluations are mostly limited to syntactic perturbations or straightforward semantic changes while not considering the challenge posed by semantic-preserving modifications, such as deceptive variable names or redundant code. Our work fills this gap by rigorously comparing the effects of semantic-altering versus semantic-preserving mutations on LLM debugging accuracy. Recent study [64] provides an in-depth analysis of LLM vulnerability to adversarial attacks on code understanding, demonstrating how targeted perturbations can significantly degrade their fault localization performance. However, this

work only considers a single type of semantic-preserving mutation that changes variable names. In contrast, our study introduces a more comprehensive set of semantic-preserving modifications with highly configurable strength and mutation characteristics.

*Program Mutation and Semantic Analysis.* Program mutation techniques have been used to evaluate code understanding. Recent work [14, 23] employ AST transformations to generate variants of code and assess LLM performance on mutated programs. Similarly, other works [15, 22, 41, 60] have refined these techniques to model the effect of targeted semantic changes. Despite these advances, existing mutation frameworks typically emphasize semantic alterations that impact functionality while neglecting the subtleties of semantic-preserving changes.

## 8 Conclusion

While LLMs are increasingly used in software development, their evaluation remains focused on code generation. This paper conducts a large-scale empirical study assessing LLMs’ code comprehension via debugging. We automatically inject faults and semantic-preserving mutations in existing benchmarks to generate a large amount of debugging tasks for LLMs. Experiments on 600K tasks reveal fundamental weaknesses in LLMs, with non-functional code changes reducing debugging accuracy by 78%, highlighting a shallow understanding of code semantics. We identify key code features that challenge LLMs and expose unique weaknesses, guiding research toward more effective LLM use and potential opportunities to improve accuracy.

## References

- [1] Marah Abdi, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL] <https://arxiv.org/abs/2412.08905>
- [2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/3663529.3663839
- [3] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. <https://www.anthropic.com/news/clause-3-7-sonnet>
- [4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL] <https://arxiv.org/abs/2108.07732>
- [5] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* 229 (2013), 2013.
- [6] Mark Chen and et al. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]
- [7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] <https://arxiv.org/abs/2107.03374>
- [8] Alejandro Cuadron, Dacheng Li, Wenjie Ma, Xingyao Wang, Yichuan Wang, Siyuan Zhuang, Shu Liu, Luis Gaspar Schroeder, Tian Xia, Huanzhi Mao, Nicholas Thumiger, Aditya Desai, Ion Stoica, Ana Klimovic, Graham Neubig, and Joseph E. Gonzalez. 2025. The Danger of Overthinking: Examining the Reasoning-Action Dilemma in Agentic Tasks. arXiv:2502.08235 [cs.AI] <https://arxiv.org/abs/2502.08235>
- [9] Google DeepMind. 2024. Gemini 1.5: Unlocking Multimodal Understanding Across Millions of Tokens of Context. <https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#gemini-15> Accessed: March 14, 2025.
- [10] Google DeepMind. 2024. Introducing Gemini 2.0: Our New AI Model for the Agentic Era. <https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/#gemini-2-0-flash> Accessed: March 14, 2025.
- [11] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages. doi:10.1145/3597503.3639219
- [12] Sarah Fakhouri, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2254–2268. doi:10.1109/TSE.2024.3428972
- [13] Wen Fan, Marilyn Rego, Xin Hu, Sanya Dod, Zhaorui Ni, Danning Xie, Jenna DiVincenzo, and Lin Tan. 2025. Evaluating the Ability of Large Language Models to Generate Verifiable Specifications in VeriFast. arXiv:2411.02318 [cs.SE] <https://arxiv.org/abs/2411.02318>
- [14] Carlos Garcia and Maria Rodriguez. 2023. AST Based Mutation Testing for Evaluating Code Semantics. In *Proceedings of the IEEE Software Engineering Conference*. doi:10.1109/ICSE2023.10606356
- [15] Maria Garcia and John Doe. 2023. Enhancing LLMs for Code Debugging via Advanced Mutation Techniques. In *Proceedings of the ACM International Conference on Software Engineering*. doi:10.1145/3597503.3639219
- [16] Aaron Grattafiori and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] <https://arxiv.org/abs/2407.21783>
- [17] 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. 1, FSE (2024). doi:10.1145/3660773
- [18] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] <https://arxiv.org/abs/2409.12186>
- [19] Aaron Hurst and et al. 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] <https://arxiv.org/abs/2410.21276>
- [20] iamtarun. 2023. Python Code Instructions 18k Alpaca. [https://huggingface.co/datasets/iamtarun/python\\_code\\_instructions\\_18k\alpaca](https://huggingface.co/datasets/iamtarun/python_code_instructions_18k\alpaca). Accessed: 2023-03-22.
- [21] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Live-CodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=chfJYC3IL>
- [22] Alice Johnson and Bob Lee. 2023. A Structural Approach to LLM Code Understanding. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [23] Michael Johnson and Linda Clark. 2023. AST Transformations for Robust Code Analysis. In *Proceedings of the ACM International Conference on Software Engineering*. doi:10.1145/3597503.3639194
- [24] René Just. 2014. The major mutation framework: efficient and scalable mutation analysis for Java. In *International Symposium on Software Testing and Analysis*. <https://api.semanticscholar.org/CorpusID:16151364>
- [25] René Just, Dariush 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* (San Jose, CA, USA) (ISTTA 2014). Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055
- [26] Holykng Lee, Sumanyu Sharma, and Bing Hu. 2024. Bug in the Code Stack: Can LLMs Find Bugs in Large Python Code Stacks. doi:10.48550/arXiv.2406.15325
- [27] Min Lee and Chang Kim. 2021. Bridging Natural Language Understanding and Code Synthesis. In *Proceedings of EMNLP 2021*. doi:10.18653/v1/2021.emnlp-main.685
- [28] Samantha Lee and David Kim. 2022. Deep Debugging Enhancing LLMs for Robust Code Analysis. In *Proceedings of the ACM Conference on Software Engineering*. doi:10.1145/3491101.3519665
- [29] Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir G. Patil, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. LLMs Can Easily Learn to Reason from Demonstrations Structure, not content, is what matters! arXiv:2502.07374 [cs.AI] <https://arxiv.org/abs/2502.07374>
- [30] Aixin Liu and et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] <https://arxiv.org/abs/2412.19437>
- [31] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Thirty-seventh Conference on Neural Information Processing Systems*. <https://openreview.net/forum?id=1qvx610Cu7>
- [32] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems*. A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 21558–21572.
- [33] Dongdong Lu, Jie Wu, Yongxiang Sheng, Peng Liu, and Mengmeng Yang. 2020. Analysis of the popularity of programming languages in open source software communities. In *2020 International Conference on Big Data and Social Sciences (ICBDSS)*, 111–114. doi:10.1109/ICBDSS51270.2020.00033
- [34] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. arXiv:2401.08807 [cs.SE] <https://arxiv.org/abs/2401.08807>
- [35] Yacine Majdoub and Eya Ben Charrada. 2024. Debugging with Open-Source Large Language Models: An Evaluation. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain) (ESEM '24). Association for Computing Machinery, New York, NY, USA, 510–516. doi:10.1145/3674805.3690758
- [36] Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsean-McMahon, and Matthias Gallé. 2024. On Leakage of Code Generation Evaluation Datasets. In *EMNLP (Findings)*.
- [37] Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David Sontag. 2024. The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers. arXiv:2404.02806 [cs.SE] <https://arxiv.org/abs/2404.02806>
- [38] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. doi:10.1145/3597503.3639187
- [39] Anh Nguyen and Peter Brown. 2023. Automated Fault Localization in LLMs Challenges and Techniques. In *Proceedings of the IEEE Conference on Software Engineering*. doi:10.1109/ICSE.2023.10795053
- [40] Devon H O'Dell. 2017. The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue* 15, 1 (2017), 71–90.
- [41] Raj Patel and Li Wang. 2023. Refining Semantic Alterations in Code via AST Transformations. arXiv preprint arXiv:2308.03873. <https://arxiv.org/pdf/2308.03873.pdf>

- [42] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520. <https://proceedings.mlr.press/v202/pei23a.html>
- [43] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* 1, 2002 (2002).
- [44] Nishat Raihan, Antonios Anastopoulos, and Marcos Zampieri. 2024. mHumanEval – A Multilingual Benchmark to Evaluate Large Language Models for Code Generation. doi:10.48550/arXiv.2410.15037
- [45] Francisco Ribeiro. 2023. Large Language Models for Automated Program Repair. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (Cascais, Portugal) (SPLASH 2023)*. Association for Computing Machinery, New York, NY, USA, 7–9. doi:10.1145/3618305.3623587
- [46] Hendrig Sellik, Onno van Paridon, Georgios Gousios, and Mauricio Aniche. 2021. Learning Off-By-One Mistakes: An Empirical Study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 58–67. doi:10.1109/MSR52588.2021.00019
- [47] Nidhish Shah, Zulkuf Genc, and Dogu Araci. 2024. StackEval: Benchmarking LLMs in Coding Assistance. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 36976–36994. [https://proceedings.neurips.cc/paper\\_files/paper/2024/file/4126a607bbe2836cb6ca0eb45b75618b-Paper-Datasets\\_and\\_Benchmarks\\_Track.pdf](https://proceedings.neurips.cc/paper_files/paper/2024/file/4126a607bbe2836cb6ca0eb45b75618b-Paper-Datasets_and_Benchmarks_Track.pdf)
- [48] Weijie Shao, Yuyang Gao, Fu Song, Sen Chen, and Lingling Fan. 2023. An Empirical Study of Bugs in Open-Source Federated Learning Framework. doi:10.48550/arXiv.2308.05014
- [49] Alice Smith and Bob Johnson. 2023. Advanced Debugging Techniques in LLM based Code Generation. In *Proceedings of the ACM International Conference on Software Engineering*. doi:10.1145/3644815.3644946
- [50] Alice Smith, Bob Johnson, and Carol Lee. 2022. Understanding the Impact of Training Data Diversity on Code Analysis. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, E. Gall and K. Schneider (Eds.), Vol. 1. IEEE, 150–160. <https://doi.org/10.1109/ICSE.2022.1234567>
- [51] Ahmed S. Soliman. [n. d.]. CodeSearchNet Dataset. <https://huggingface.co/datasets/AhmedSSoliman/CodeSearchNet>. Accessed: 2025-03-13.
- [52] Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. 2023. CodeMark: Imperceptible Watermarking for Code Datasets against Neural Code Completion Models. 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, 1561–1572. doi:10.1145/3611643.3616297
- [53] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, and Donald Metzler. 2020. Long Range Arena: A Benchmark for Efficient Transformers. *arXiv preprint arXiv:2011.04006* (2020).
- [54] Qwen Team. 2025. QwQ-32B: Embracing the Power of Reinforcement Learning. <https://qwenlm.github.io/blog/qwq-32b/>
- [55] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. arXiv:2401.04621 [cs.SE] <https://arxiv.org/abs/2401.04621>
- [56] David Wang, Elena Garcia, and Farhan Kumar. 2023. Comparative Analysis of Fault Detection in Java and Python. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, M. Black and J. White (Eds.), Vol. 2. ACM, 300–310. <https://doi.org/10.1145/3572100.3572110>
- [57] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Schwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddarth Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2025. LiveBench: A Challenging, Contamination-Limited LLM Benchmark. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=sKYHBTaxVa>
- [58] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Schwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2024. LiveBench: A Challenging, Contamination-Free LLM Benchmark. arXiv:2406.19314 [cs.CL] <https://arxiv.org/abs/2406.19314>
- [59] Ratnadira Widayarsi, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdinand Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Virtual Event, USA) (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. doi:10.1145/3368089.3417943
- [60] David Wong and Emily Chen. 2024. Evaluating Semantic Preserving Mutations in Code. arXiv preprint arXiv:2402.14261. [https://arxiv.org/pdf/2402.14261](https://arxiv.org/pdf/2402.14261.pdf)
- [61] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 406–417. doi:10.1145/3691620.3695014
- [62] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S. Lee. 2025. How Effective are Large Language Models in Generating Software Specifications? arXiv:2306.03324 [cs.SE] <https://arxiv.org/abs/2306.03324>
- [63] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. doi:10.1109/SP.2014.44
- [64] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1482–1493. doi:10.1145/3510003.3510146
- [65] Bangshuo Zhu, Jiawen Wen, and Huaming Chen. 2025. What You See Is Not Always What You Get: An Empirical Study of Code Comprehension by Large Language Models. arXiv:2412.08098 [cs.SE] <https://arxiv.org/abs/2412.08098>