Large Language Models Are Not (Yet) Robust in Understanding Code Against Semantics-Preserving Mutations

Pedro Orvalho[1⋆][0000−0002−7407−5967] and Marta Kwiatkowska[2][0000−0001−9022−7599]

[1] Artificial Intelligence Research Institute (IIIA), Consejo Superior de Investigaciones Científicas (CSIC), Barcelona, Catalonia, Spain pedro.orvalho@iiia.csic.es
[2] Department of Computer Science, University of Oxford, Oxford, UK

Abstract. With the widespread adoption of *vibe coding*, understanding the reasoning and robustness of Large Language Models (LLMs) is critical for their reliable use in programming tasks. While recent studies assess LLMs' ability to predict program outputs, most focus on accuracy alone, without evaluating the underlying reasoning. Moreover, it has been observed on mathematical reasoning tasks that LLMs can arrive at correct answers through flawed logic, raising concerns about similar issues in code understanding. In this paper we assess whether state-of-the-art LLMs can reason about Python programs or are simply guessing. We apply five semantics-preserving code mutations: renaming variables, mirroring comparison expressions, swapping if-else branches, converting `for` loops to `while`, and loop unrolling. These mutations maintain program semantics while altering its syntax. We evaluated nine LLMs, including both open-source and closed-access models, and performed a human expert analysis using LiveCodeBench to assess whether correct predictions are based on sound reasoning. We also evaluated prediction stability across different code mutations on LiveCodeBench and CruxEval. While proprietary models achieve the strongest predictive accuracy and reasoning quality in the expert evaluation, our robustness analysis reveals substantial fragility under semantics-preserving transformations. Our findings show that the evaluated code-specialised LLMs produce correct predictions based on flawed reasoning in up to 45% of cases. Furthermore, LLMs often change predictions in response to our code mutations, with performance drops reaching up to 70%, indicating that they do not yet exhibit stable, semantically grounded reasoning, even when initial accuracy is high.

Keywords: LLMs for Code Understanding · Semantic Robustness of Large Language Models · Code Mutations.

⋆ Corresponding Author. PO is now affiliated with IIIA-CSIC, Spain. This work was carried out while he was at the University of Oxford.

1 Introduction

Large Language Models (LLMs) have rapidly become integral to a wide range of daily tasks, from writing assistance to code generation. In particular, the software development community has embraced LLM-based tools, such as GitHub Copilot and ChatGPT, to streamline code workflows, assist in debugging, and even automate code completion and review [3,11]. These tools are widely used, and often blindly, with developers placing significant trust in their capabilities [15]. However, this growing reliance on LLMs for coding tasks raises a fundamental question: *To what extent do LLMs truly understand code and the underlying semantics of programs?*

While recent LLMs can produce syntactically correct code and even solve competitive programming problems, there is a risk that their responses may reflect pattern recognition over code syntax rather than genuine semantic understanding [19]. Although LLMs rely on statistical associations to generate output, their ability to mimic reasoning can obscure the limitations of such models, potentially leading to an overestimation of their reliability in critical development contexts [6]. Furthermore, based on the growing dependence of programmers on LLMs, several works in the past year have analysed LLMs' reasoning about code through output prediction tasks [3,7,9]. Some LLMs, such as SemCoder [3], were trained to *understand* program semantics through a monologue reasoning strategy (i.e., Chain-of-Thought), where the model verbally simulates both the high-level syntax and low-level execution effects of code. However, some recent work has shown that in other domains, such as mathematical competitions [19], LLMs tend to provide accurate predictions, but based on flawed reasoning.

In this paper, we evaluate LLMs on the task of reasoning about code and predicting program outputs for given inputs, using two well-studied benchmarks: LiveCodeBench [9] and CruxEval [7]. We define *sound semantic reasoning* in LLMs in terms of their reasoning traces being semantically accurate, that is, when the models predict the correct output and correctly describe the control and data flow that takes the program from the inputs to the expected output. First, we conduct a manual expert evaluation on LiveCodeBench to determine whether correct predictions are derived from logically sound reasoning, flawed reasoning, or mere guesses. Second, we assess model robustness by applying semantics-preserving code mutations to both benchmarks and analysing output prediction consistency. These mutations, which are small syntactic modifications that preserve runtime behaviour, test whether LLMs' understanding of code is robust to syntactic changes. We apply five semantics-preserving code mutations explained in Section 2: (1) renaming variables, (2) mirroring comparison expressions, (3) swapping if-else branches, (4) converting `for` loops to `while` loops, and (5) unrolling the final iterations of loops. These code transformations allow us to evaluate whether LLMs are sensitive to syntax or capable of reasoning about code semantics.

Our experiments consider nine state-of-the-art LLMs, including five code-specialised open-weight models (CodeGemma, GraniteCode, Qwen2.5-Coder, Qwen3-Coder, and SemCoder), two proprietary coding assistants (GPT-

-5.2 and Gemini-3), and the general-purpose models, Mistral and Llama3.2, for comparison. Our expert analysis shows that LLMs frequently produce correct predictions despite flawed reasoning, affecting up to 55% of correct predictions for Llama3.2 and 45% for code-specialised models. Interactive querying improves both prediction accuracy and reasoning quality, increasing the proportion of sound reasoning by up to 14% for code models and 19% for Llama3.2.

Our robustness evaluation further shows that LLMs frequently change their predictions under semantics-preserving code mutations, with performance drops of up to 70%. Variable renaming and loop unrolling are particularly challenging, indicating that current models remain sensitive to superficial syntactic variation despite unchanged program semantics.

Among the evaluated models, GPT-5.2 and Gemini-3 achieve the highest prediction accuracy and reasoning quality, whilst Qwen3-Coder is the strongest open-weight model. However, even these models exhibit substantial performance degradation under semantics-preserving transformations, suggesting that high benchmark accuracy does not necessarily imply semantic robustness. These results should also be interpreted with caution, as GPT-5.2, Gemini-3, and Qwen3-Coder were released after LiveCodeBench and may therefore have been trained on overlapping data. Overall, our findings suggest that current code-oriented LLMs remain brittle under semantics-preserving transformations and that improving semantic robustness will likely require approaches that go beyond simply scaling models or training on additional code.

In summary, this paper makes the following contributions:

- We perform a manual expert evaluation to assess whether LLMs' code output predictions are based on logically sound code reasoning, flawed reasoning, or mere guesses.
- We evaluate LLMs' output prediction stability across five different semantics-preserving code mutations.
- Experiments show that LLMs frequently alter their predictions when subjected to semantics-preserving code mutations, highlighting that they do not yet exhibit stable, semantically grounded reasoning.
- Our code is publicly available on GitHub: `https://github.com/pmorvalho/EPIA26-LLMCs-Semantic-Robustness`.

2 Semantics-Preserving Code Mutations

We introduce a set of semantics-preserving program transformations designed to syntactically modify Python programs without altering their semantics, i.e., their runtime behaviour. Let us first define semantics-preserving program mutations.

Definition 1 (Semantics-Preserving Code Mutation (SPCM)). *A* SPCM *is a syntactic transformation to program P that generates a new program P_m by syntactically replacing a subset S_1 of P's statements ($S_1 \subseteq P$) with another set of statements S_2, such that $P_m = ((P \backslash S_1) \cup S_2)$, and P_m is also semantically consistent with a given specification (i.e., test suite): $\forall (t^i_{in}, t^i_{out}) \in T \ : \ P_m(t^i_{in}) = t^i_{out}$.*

Algorithm 1 Function `minPossSum(n, target)`.

```
def minPossSum(n:int, t:int)->int:
    i = 1
    arr = {1}
    while len(arr) < n:
        i += 1
        if t - i not in arr:
            arr.add(i)
    return sum(arr)
```

Algorithm 2 Algorithm 1 after renaming variable `arr`.

```
def minPossSum(n:int, t:int)->int:
    i = 1
    eAJMfVcq = {1}
    while len(eAJMfVcq) < n:
        i += 1
        if t - i not in eAJMfVcq:
            eAJMfVcq.add(i)
    return sum(eAJMfVcq)
```

These code mutations are essential for tasks such as testing the robustness of code understanding models and augmenting training data in a semantics-preserving manner. In this section, we describe five mutations implemented in our work: variable renaming, comparison mirroring, swap if-else statements, for-to-while loop conversion, and partial loop unrolling. Some of these code mutations have been previously employed [8,17] to augment benchmarks with semantically equivalent, yet syntactically different, versions of the original programs. Please refer to our extended version [18] for more details about the code mutations used in our work.

Variable Renaming. This program transformation systematically renames local variables, function arguments, or function names using fresh identifiers that do not conflict with existing symbols or Python built-ins. A consistent mapping is maintained within each scope to ensure correctness. This program mutation preserves the program's semantics while altering its lexical structure. Algorithm 2 illustrates the result of applying the variable-renaming transformation to Algorithm 1, replacing `arr` with `eAJMfVcq`.

Comparison Expression Mirroring. This transformation mirrors comparison expressions by swapping operands and applying their logically equivalent inverse operators. This mutation preserves program semantics and applies to all symmetric and reversible binary comparisons. It is particularly useful for assessing models that rely on syntax, such as token order or abstract syntax tree structure, for reasoning about code.

Swap If-Else Statements. Another semantics-preserving mutation we use is the swapping of if-else statements, in which the `if` and `else` blocks are swapped and the `if` condition is logically negated. For example, a condition `if x > 0:` is rewritten as `if not (x > 0):`, with the bodies of the `if` and `else` blocks swapped accordingly. This code mutation maintains the program's behaviour but alters the logical structure and control flow graph. Robust models should recognise the semantic equivalence of these logically inverted blocks and produce consistent output predictions, regardless of the branching structure.

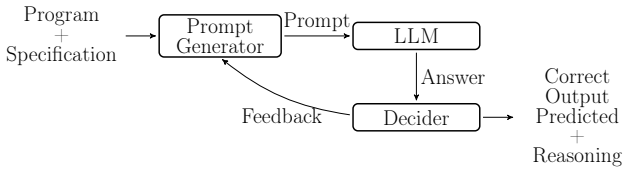

Fig. 1. LLM-Based Program Output Prediction.

For-to-While Loop Conversion. This code transformation rewrites `for` loops into semantically equivalent `while` loops. It introduces an index variable and manually iterates over the collection using the `len()` function and explicit indexing, while preserving all loop control logic. This transformation is particularly useful for assessing the robustness of LLMs in recognising semantically equivalent loop constructs.

Partial Loop Unrolling. To simulate partial loop unrolling while preserving semantics, we extract the last one or two iterations of a `while` loop body and duplicate them after the loop. The loop condition is also modified to run fewer iterations, e.g., reducing a bound `n` to `n - 1` in a `while i < n` condition. The extracted iterations are then executed sequentially after the loop, preserving the program behaviour.

3 LLM-Based Program Output Prediction

As illustrated in Figure 1, we address the task of using LLMs to predict the output of a Python program for a given input by employing an iterative querying strategy. We chose code output prediction as our evaluation task because success in this task strongly correlates with "understanding" of code semantics, but it is also easy to check and automate the interaction. Starting with a program and its input-output specification, we invoke a prompt generator that constructs and submits the query to the LLM. We then evaluate whether the predicted output matches the expected one. If the prediction is incorrect, a feedback prompt is generated and sent back to the LLM, explicitly requesting a revised answer. This loop continues until the model produces the correct output, the time limit is reached, or the number of iterations exceeds five, typically indicating that the model is stuck and repeatedly returning the same incorrect answer. Please refer to our extended version [18] for all the prompt templates used in the various interactions with the LLMs.

Prompts. The prompts used to query the LLMs follow a similar format to those adopted in prior works [3,7]. Each prompt asks the model to complete a Python assertion, given the function signature and a test input. To guide the model's response format, we include a brief example demonstrating how the answer should be wrapped within specific tags.

Feedback. If the output predicted by the LLM is incorrect, i.e., it does not match the expected output from the test suite, we provide feedback to the model

through iterative querying; this does not apply to two of the evaluated models (SemCoder and Mistral) that do not support more than one interaction. Specifically, a follow-up prompt is sent indicating that the previous response was incorrect. This feedback aims to simulate a correction loop and assess whether the LLM can refine its prediction after being notified of a mistake.

4 Experiments

The goal of our experiments was to answer the following research questions (RQs): **RQ1.** Are Large Language Models (LLMs) truly reasoning about code semantics, or merely guessing likely answers? **RQ2.** Does the interactive querying process help LLMs arrive at correct predictions supported by logically sound reasoning? **RQ3.** Are LLMs robust in understanding code against semantics-preserving mutations? Our objective is to evaluate LLMs on the task of reasoning about code and predicting program outputs for given inputs. All experiments were conducted with a 5-minute time limit to bound each interaction loop described in Section 3.

4.1 Experimental Setup

Evaluation Benchmarks: We evaluate our approach on two widely used Python benchmarks: LiveCodeBench [9] and CruxEval [7]. We use the code execution split of LiveCodeBench, comprising 479 programs collected from programming competition platforms such as `LeetCode` [10]. CruxEval comprises 800 Python functions generated by CodeLlama [2], each paired with input–output examples.

Both benchmarks are widely used to evaluate LLMs on code reasoning tasks, including program output prediction, synthesis, and repair. We apply each semantics-preserving mutation described in Section 2 by randomly selecting an applicable statement in each program. To ensure correctness, every mutated program is validated against the original benchmark assertions, and only semantics-preserving transformations are retained. For each mutation type, we generate a single mutated variant per program containing at most one transformation. Further implementation details are available in our publicly released code.

Large Language Models (LLMs): We evaluate nine different LLMs: seven open-access LLMs using the iterative querying setup described in Section 3, CodeGemma [1] (7B), GraniteCode [5] (8B), Mistral [14] (7B), Qwen2.5-Coder [20] (7B), Qwen3-Coder [21] (30B), SemCoder [3] (7B), and Llama3.2 [12] (3B), and two closed-access models, OpenAI's GPT-5.2 [16] and Google's Gemini-3 [4], which are among the most widely adopted state-of-the-art proprietary models. We evaluated the proprietary models through their official APIs. As their internal implementations are not publicly documented, we cannot exclude the possibility that they rely on additional inference mechanisms or system-level components unavailable to open-weight models. Consequently, comparisons should be interpreted with this limitation. Moreover, all models were run with temperature 0 to eliminate variance across different runs.

LLMs	CodeGemma	GraniteCode	Qwen2.5-Coder	Qwen3-Coder	Mistral	SemCoder	Llama3.2	GPT-5.2	Gemini-3
%Failed Predict.	66.8	65.1	37.4	49.5	67.4	52.0	61.4	2.5	0.0
%Correct Predict.	33.2	34.9	62.6	50.5	32.6	48.0	38.6	97.5	100.0
%Correct Guesses based on flawed reasoning	44.7	43.1	10.3	3.0	50.0	16.1	55.1	0.2	0.2
%Correct Predict. based on sound reasoning (>1 iteration)	3.8	13.8	8.0	1.6	–	–	18.9	8.6	0.0
%Correct Predict. based on sound reasoning (=1 iteration)	51.6	43.1	81.7	95.4	50.0	83.9	25.9	91.2	99.8

Table 1. In-depth Analysis of LLMs' Reasoning on LiveCodeBench.

4.2 Expert Analysis of LLMs' Reasoning About Code

To answer our first research question (**RQ1**), Table 1 presents a detailed expert analysis of the reasoning behaviour of the evaluated LLMs on LiveCodeBench, focusing on prediction outcomes and code understanding. The lower section of Table 1 reports the proportion of correct predictions that were either guesses based on flawed reasoning or grounded in sound semantic reasoning, distinguishing between first-attempt success and success achieved after iterative refinement.

We define *sound semantic reasoning* as reasoning traces that are semantically accurate, that is, when models not only predict the correct output but also correctly describe the control and data flow leading from the inputs to the expected output. However, LLMs are autoregressive probabilistic models, and correct predictions may arise even when intermediate reasoning traces are imperfect. All interactions between the evaluated LLMs and programs in LiveCodeBench were manually inspected. Labelling was conducted blind to model identity. Each chain-of-thought (CoT) response leading to a correct prediction was independently labelled three times by experts with strong programming backgrounds, and disagreements were subsequently resolved. We did not extend this in-depth analysis to CruxEval due to its substantially larger size.

Overall predictive performance. Among open-access models, Qwen2.5-Coder achieves the highest rate of correct predictions (62.6%), followed by Qwen3-Coder (50.5%) and SemCoder (48.0%). In contrast, the closed-access models substantially outperform all open-access models, with GPT-5.2 achieving 97.5% and Gemini-3 100.0% correct predictions.

Reasoning quality and guessing behaviour. The proprietary models combine very high accuracy with negligible reliance on flawed reasoning. Only 0.2% of correct predictions from both GPT-5.2 and Gemini-3 are classified as guesses. Moreover, the overwhelming majority of their correct predictions are grounded in sound reasoning within a single iteration, 91.2% for GPT-5.2 and 99.8% for Gemini-3. Among open-access models, reasoning robustness varies considerably. Qwen2.5-Coder exhibits a relatively low guessing rate (10.3%) and a high share of correct predictions grounded in sound reasoning within a single iteration (81.7%), indicating comparatively stable reasoning when successful.

SemCoder shows a similarly high proportion of sound reasoning when correct (83.9%), though with lower overall predictive accuracy. Notably, Qwen-3-Coder demonstrates a very high proportion of sound first-iteration reasoning (95.4%) and a low guessing rate (3.0%), suggesting that when it predicts correctly, its reasoning is typically coherent and semantically grounded. However, its overall predictive accuracy remains substantially below that of the closed-access models. In contrast, Llama3.2, Mistral, CodeGemma, and GraniteCode exhibit higher guessing rates, with 55.1%, 50.0%, 44.7%, and 43.1% of correct predictions, respectively, classified as flawed reasoning, indicating a greater tendency to reach correct outputs without fully sound logical justification.

Impact of interactive querying (RQ2). To answer RQ2, we measure the share of correct predictions obtained through sound reasoning after an initial failure. Interactive querying helps some open-access models: Llama3.2, GraniteCode, and Qwen2.5-Coder recover 18.9%, 13.8%, and 8.0% of their correct predictions, respectively, while CodeGemma and Qwen3-Coder gain only 3.8% and 1.6%. This suggests that iterative feedback mainly helps weaker models recover from initial mistakes. For closed-access models, gains are limited: GPT-5.2 attributes 8.6% of its correct predictions to later refinement, whereas Gemini-3 shows no measurable improvement, consistent with its near-perfect first-attempt accuracy.

Release timeline and potential benchmark contamination. These results must be interpreted in light of model release timelines. The evaluated versions of GPT-5.2 and Gemini-3 correspond to releases from the end of 2025, whereas LiveCodeBench was published in 2024. All other evaluated models, except Qwen3-Coder, were released prior to the publication of LiveCodeBench. Consequently, Qwen3-Coder, GPT-5.2, and Gemini-3 may have had access to LiveCodeBench or overlapping data during training. Although the exact composition of their training corpora is unknown, it is possible that part of their strong performance is attributable to prior exposure to benchmark instances. Therefore, the near-perfect results of GPT-5.2 and Gemini-3, and the strong reasoning quality of Qwen3-Coder, should be interpreted with caution.

Cost and accessibility considerations. Lastly, performance differences should be considered alongside accessibility. All evaluated open-access models are freely available, whereas GPT-5.2 and Gemini-3 are proprietary systems requiring paid API access. Although the closed-access models achieve substantially higher predictive accuracy and reasoning robustness, this advantage comes with a non-trivial cost differential that may limit adoption in academic or resource-constrained settings.

4.3 Robustness to Semantics-Preserving Mutations

Tables 2, and 3 provide insights into the performance of the evaluated LLMs when predicting outputs on the original benchmarks, LiveCodeBench and

LLMs	Original Code	LC	EM	VR	IE	LU
CodeGemma	33.2%	26.3 (-6.9)	31.7 (-1.5)	38.0 (+4.8)	28.4 (-4.8)	10.9 (-22.3)
GraniteCode	34.9%	27.1 (-7.7)	31.3 (-3.5)	38.8 (+4.0)	29.2 (-5.6)	8.4 (-26.5)
Llama3.2	38.6%	34.4 (-4.2)	34.2 (-4.4)	46.8 (+8.1)	30.1 (-8.6)	9.4 (-29.2)
Mistral	32.6%	24.0 (-8.6)	29.2 (-3.3)	38.2 (+5.6)	27.1 (-5.4)	10.4 (-22.1)
Qwen2.5-Coder	62.6%	49.1 (-13.6)	58.5 (-4.2)	76.6 (+14.0)	50.7 (-11.9)	18.4 (-44.3)
Qwen3-Coder	50.5%	37.8 (-12.7)	44.1 (-6.5)	54.7 (+4.2)	33.2 (-17.3)	15.4 (-35.1)
SemCoder	48.0%	40.1 (-7.9)	48.9 (+0.8)	62.4 (+14.4)	40.3 (-7.7)	15.2 (-32.8)
GPT-5.2	97.1%	72.4 (-24.6)	81.0 (-16.1)	98.5 (+1.5)	78.3 (-18.8)	29.0 (-68.1)
Gemini-3	100.0%	55.3 (-44.7)	76.0 (-24.0)	68.1 (-31.9)	71.4 (-28.6)	29.6 (-70.4)

Table 2. Correct prediction rate of each LLM on LiveCodeBench when applying different code mutations: loop conversion (LC), expression mirroring (EM), variable renaming (VR), swap if-else statments (IE), and loop unrolling (LU).

CruxEval, and after applying semantics-preserving code mutations (see Section 2). Since these transformations preserve program behaviour while altering syntactic structure, they enable a controlled evaluation of model robustness and semantic understanding.

Tables 2 and 3 report the percentage of distinct programs from LiveCodeBench and CruxEval, respectively, for which each LLM produces a correct output prediction under different code mutations. These rates reflect the ability of each model to handle syntactic variation without semantic degradation. Performance varies substantially across mutation types and across models. On LiveCodeBench, variable renaming consistently improves accuracy for most open-access models. For instance, Qwen2.5-Coder improves from 62.6% to 76.6% (+14.0), SemCoder from 48.0% to 62.4% (+14.4), and Qwen3-Coder from 50.5% to 54.7% (+4.2). Even GPT-5.2 shows a small improvement (+1.5), although Gemini-3 exhibits a notable drop under this mutation (from 100.0% to 68.1%). In contrast, structurally more invasive mutations such as loop unrolling cause dramatic performance degradation across all models. On LiveCodeBench, Qwen2.5-Coder drops by 44.3 points, Qwen3-Coder by 35.1, SemCoder by 32.8, GPT-5.2 by 68.1, and Gemini-3 by 70.4. Loop conversion and swapping `if-else` branches similarly produce large declines, particularly for the closed-access models. For example, Gemini-3 drops by 44.7 points under loop conversion and 28.6 points under branch swapping.

A similar pattern is observed on CruxEval (see Table 3). While baseline accuracies are generally slightly lower than on LiveCodeBench, mutation-induced drops remain substantial. Qwen2.5-Coder decreases by 30.6 points under loop conversion and 43.5 under loop unrolling. Qwen3-Coder and SemCoder show comparable declines. GPT-5.2 drops by 43.8 and 63.0 points under loop conversion and loop unrolling, respectively, while Gemini-3 decreases by 47.6 and 52.4 points under the same mutations. These results indicate that even models with near-perfect baseline performance, such as GPT-5.2 and Gemini-3, are highly sensitive to semantics-preserving structural rewrites. Overall, model performance remains strongly dependent on syntactic form, suggesting limited robustness to semantics-preserving code transformations.

LLMs	Original Code	LC	EM	VR	IE	LU
CodeGemma	30.9%	15.8 (-15.1)	17.6 (-13.3)	34.9 (+4.0)	20.1 (-10.8)	8.9 (-22.0)
GraniteCode	32.6%	14.2 (-18.4)	17.1 (-15.5)	35.4 (+2.8)	19.8 (-12.9)	8.1 (-24.5)
Llama3.2	26.5%	15.0 (-11.5)	17.4 (-9.1)	37.5 (+11.0)	19.2 (-7.2)	6.6 (-19.9)
Mistral	23.8%	11.0 (-12.8)	13.4 (-10.4)	25.5 (+1.8)	13.9 (-9.9)	6.5 (-17.2)
Qwen2.5-Coder	59.6%	29.0 (-30.6)	36.1 (-23.5)	64.9 (+5.3)	33.5 (-26.1)	16.1 (-43.5)
Qwen3-Coder	48.9%	23.1 (-25.8)	26.2 (-22.6)	49.9 (+1.0)	26.2 (-22.6)	12.5 (-36.4)
SemCoder	50.6%	25.2 (-25.4)	29.9 (-20.8)	55.8 (+5.1)	30.4 (-20.2)	12.8 (-37.9)
GPT-5.2	86.0%	42.2 (-43.8)	47.9 (-38.1)	87.0 (+1.0)	52.9 (-33.1)	23.0 (-63.0)
Gemini-3	76.8%	29.1 (-47.6)	50.9 (-25.9)	66.2 (-10.5)	41.2 (-35.5)	24.4 (-52.4)

Table 3. Correct prediction rate of each LLM on CruxEval when applying different code mutations: loop conversion (LC), expression mirroring (EM), variable renaming (VR), swap if-else statments (IE), and loop unrolling (LU).

Addressing **RQ3**, we conclude that current LLMs do not yet demonstrate stable, semantically grounded robustness. A truly robust model would exhibit minimal performance variation across equivalent program rewrites. However, the considerable fluctuations observed across both benchmarks indicate that predictions remain heavily influenced by surface-level syntactic cues rather than invariant program semantics. Robustness to such benign transformations is essential for reliable, generalisable, and explainable code reasoning systems.

5 Related Work

In recent years, several works have explored the ability of code models to *reason semantically* about programs, primarily through tasks such as code generation and output prediction [3,7,8,9,23,25]. Prior work improved the robustness of code models by introducing semantics-preserving code transformations, such as adding dead code, renaming variables, or inserting print statements, that preserve program semantics whilst fooling smaller code models (e.g., seq2seq and code2seq) [8]. These transformations were primarily used to construct adversarial examples and to enable robustness-aware training. More recently, Spiess et al. [23] revisited semantics-preserving transformations in the context of modern LLMs, demonstrating that even frontier models exhibit substantial performance degradation under both program transformations and input perturbations, despite achieving near-perfect accuracy on the original benchmark. Their work further analyses robustness with respect to runtime exceptions and execution-path complexity, highlighting that benchmark accuracy alone does not necessarily reflect robust execution reasoning. Our work complements these findings by focusing on a complementary aspect of semantic reasoning: rather than studying robustness solely through predictive performance, we combine robustness evaluation with a manual expert assessment of Chain-of-Thought reasoning, allowing us to distinguish correct predictions supported by semantically sound reasoning from those produced through flawed or superficial reasoning. Furthermore, unlike previous work, we evaluate robustness across both LiveCodeBench and CruxEval and investigate the effect of iterative interaction on reasoning quality.

ReCode [25] evaluates the robustness of code generation models by applying diverse code perturbations to prompts and quantifying the resulting accuracy degradation, thereby providing post-hoc measures of model stability. Unlike ReCode, which focuses on code generation, our work studies program understanding and output prediction, enabling us to investigate not only whether predictions change under semantics-preserving mutations but also whether correct predictions are supported by semantically coherent reasoning. It has also been shown that LLMs frequently generate Chain-of-Thought (CoT) explanations that do not faithfully reflect the factors driving their predictions [24]. However, that analysis was restricted to question-answering tasks rather than code. Our work extends this line of research to program understanding, showing that code-oriented LLMs likewise produce correct predictions that are often accompanied by flawed or incomplete reasoning. Recent work has further demonstrated that CoT prompting can itself be brittle, with performance degrading when models are evaluated on tasks, reasoning lengths, or prompt formats that deviate from the training distribution [26]. Unlike these studies, we investigate the relationship between reasoning quality and semantic robustness in code understanding.

In mathematical reasoning, strong benchmark performance has similarly been shown to coexist with substantial weaknesses in formal proof construction, including flawed logic and unjustified assumptions [19]. Similarly, SeqCoBench [13] introduced a benchmark comprising more than twenty Python program transformations to evaluate whether LLMs recognise functional equivalence, reporting that current models perform only marginally better than syntax-based baselines. More broadly, a recent survey [22] categorises reasoning failures in LLMs across informal, formal, and embodied domains, highlighting their sensitivity to small input perturbations. Taken together with our results, these studies suggest that robustness to superficial variations remains a general challenge for contemporary language models. Our work contributes to this growing body of evidence by demonstrating that, even for state-of-the-art code-oriented models, high predictive accuracy does not necessarily imply semantically robust or consistently sound reasoning.

6 Threats to Validity

Internal Validity. We validated every semantics-preserving mutation by executing the original benchmark assertions and verifying identical outputs. We note that Chain-of-Thought (CoT) explanations are not assumed to faithfully represent the models' internal reasoning; rather, our analysis evaluates the consistency between the generated reasoning, predicted outputs, and robustness under semantics-preserving transformations. *Construct Validity.* We operationalise semantic understanding through prediction consistency under semantics-preserving transformations and the semantic correctness of reasoning traces. Although these provide useful observable measures, they remain proxies for program understanding and do not capture all aspects of code reasoning, such as synthesis, debugging, or repair. *External Validity.* Our evaluation is limited to Python programs from

LiveCodeBench and CruxEval, and the findings may not generalise to other programming languages or larger software systems. Furthermore, proprietary models are evaluated through vendor APIs, whose internal implementations are not publicly documented. Finally, as LLMs evolve rapidly, our conclusions apply only to the evaluated model versions.

7 Conclusion

In this work, we assess the code reasoning and semantic robustness of Large Language Models (LLMs) for program output prediction, going beyond accuracy to test whether correct outputs co-occur with *sound semantic reasoning* and remain stable under semantics-preserving mutations. Using nine LLMs on LiveCodeBench, expert annotation shows that many correct answers still come from flawed or superficial reasoning. GPT-5.2 and Gemini-3 achieve the best accuracy and reasoning quality overall, with Qwen3-Coder the strongest open-access model. However, high accuracy does not guarantee robustness. Our mutation-based evaluation on LiveCodeBench and CruxEval shows strong sensitivity to minor syntactic rewrites despite unchanged semantics. Even the best proprietary models suffer drops of up to 70% under semantics-preserving transformations. These findings demonstrate that accuracy alone is not a reliable proxy for genuine code understanding, and that trustworthy software engineering applications require evaluation of both reasoning consistency and robustness to semantically equivalent program variants. While current state-of-the-art code-focused models excel at pattern recognition and surface-level syntax manipulation, our findings show that they fail to maintain stable predictions under semantics-preserving transformations, suggesting that current LLMs do not yet exhibit consistently robust semantic reasoning about code.

Acknowledgments. This work was supported by grants PID2022-139835NB-C21 and PID2025-171340NB-I00 funded by MCIN/AEI/10.13039/501100011033. This project has received funding from the European Union's HORIZON-MSCA-2025-PF research and innovation programme under the Marie Skłodowska-Curie (Sherlock4Py, GA No 101269051). This project received funding from the ERC under the European Union's Horizon 2020 research and innovation programme (FUN2MODEL, grant agreement No. 834115) and ELSA: European Lighthouse on Secure and Safe AI project (grant agreement No. 101070617 under UK guarantee).

References

1. CodeGemma: Codegemma: Open code models based on gemma. CoRR **abs/2406.11409** (2024)
2. CodeLlama: Code llama: Open foundation models for code. CoRR **abs/2308.12950** (2023)
3. Ding, Y., Peng, J., Min, M.J., Kaiser, G.E., Yang, J., Ray, B.: Semcoder: Training code language models with comprehensive semantics reasoning. In: NeurIPS (2024)

4. Google: Gemini-3 https://blog.google/products-and-platforms/products/gemini/gemini-3/ (2025), released: 2025-11-18
5. Granite: Granite code models: A family of open foundation models for code intelligence. CoRR **abs/2405.04324** (2024)
6. Gu, A., Li, W., Jain, N., Olausson, T., Lee, C., Sen, K., Solar-Lezama, A.: The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? In: ACL 2024. pp. 74–117 (2024)
7. Gu, A., Rozière, B., Leather, H., Solar-Lezama, A., Synnaeve, G., Wang, S.I.: CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. arXiv preprint arXiv:2401.03065 (2024)
8. Henkel, J., Ramakrishnan, G., Wang, Z., Albarghouthi, A., Jha, S., Reps, T.W.: Semantic robustness of models of source code. In: SANER. pp. 526–537. (2022)
9. Jain, N., Han, K., Gu, A., Li, W., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., Stoica, I.: LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. CoRR **abs/2403.07974** (2024)
10. LeetCode: https://leetcode.com (2026), accessed: 2026-05-01
11. Liang, J.T., Yang, C., Myers, B.A.: A large-scale survey on the usability of AI programming assistants: Successes and challenges. In: ICSE. (2024)
12. Llama3: The llama 3 herd of models (2024), https://arxiv.org/abs/2407.21783
13. Maveli, N., Vergari, A., Cohen, S.B.: What can large language models capture about code functional equivalence? In: NAACL (2025)
14. Mistral: Mistral 7b (2023), https://arxiv.org/abs/2310.06825
15. Oh, S., Lee, K., Park, S., Kim, D., Kim, H.: Poisoned chatgpt finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned ai models. In: 2024 IEEE Symposium on Security and Privacy (SP) (2024)
16. OpenAI: GPT5.2 https://openai.com/index/introducing-gpt-5-2/ (2025)
17. Orvalho, P., Janota, M., Manquinho, V.M.: MultIPAs: Applying Program Transformations To Introductory Programming Assignments For Data Augmentation. In: ESEC/FSE 2022. (2022)
18. Orvalho, P., Kwiatkowska, M.: Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations? CoRR **abs/2505.10443** (2025). https://doi.org/10.48550/ARXIV.2505.10443
19. Petrov, I., Dekoninck, J., Baltadzhiev, L., Drencheva, M., Minchev, K., Balunovic, M., Jovanovic, N., Vechev, M.T.: Proof or Bluff? Evaluating LLMs on 2025 USA Math Olympiad. CoRR **abs/2503.21934** (2025)
20. Qwen: Qwen2.5-coder technical report (2024), https://arxiv.org/abs/2409.12186
21. Qwen: Qwen3 technical report (2025), https://arxiv.org/abs/2505.09388
22. Song, P., Han, P., Goodman, N.: Large language model reasoning failures. Trans. Mach. Learn. Res. (2026)
23. Spiess, C., Devanbu, P., Barr, E.T.: How robustly do llms understand execution semantics? In: Proceedings of the 3rd ACM International Conference on AI-Powered Software. p. 288–298. AIware '26, ACM (2026)
24. Turpin, M., Michael, J., Perez, E., Bowman, S.R.: Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In: NeurIPS (2023)
25. Wang, S., et al: ReCode: Robustness Evaluation of Code Generation Models. In: ACL 2023. pp. 13818–13843 (2023)
26. Zhao, C., Tan, Z., Ma, P., Li, D., Jiang, B., Wang, Y., Yang, Y., Liu, H.: Is chain-of-thought reasoning of llms a mirage? A data distribution lens. CoRR **abs/2508.01191** (2025)