Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations?

Pedro Orvalho[a,*] **and Marta Kwiatkowska**[a]

[a]Department of Computer Science, University of Oxford, Oxford, UK
ORCID (Pedro Orvalho): https://orcid.org/0000-0002-7407-5967, ORCID (Marta Kwiatkowska): https://orcid.org/0000-0001-9022-7599

arXiv:2505.10443v1 [cs.SE] 15 May 2025

Abstract. Understanding the reasoning and robustness of Large Language Models (LLMs) is critical for their reliable use in programming tasks. While recent studies have assessed LLMs' ability to predict program outputs, most focus solely on the accuracy of those predictions, without evaluating the reasoning behind them. 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 work, we evaluate whether state-of-the-art LLMs with up to 8B parameters 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 six LLMs and performed a human expert analysis using LIVECODEBENCH to assess whether the correct predictions are based on sound reasoning. We also evaluated prediction stability across different code mutations on LIVECODEBENCH and CRUXEVAL. Our findings show that some LLMs, such as LLAMA3.2, produce correct predictions based on flawed reasoning in up to 61% of cases. Furthermore, LLMs often change predictions in response to our code mutations, indicating limited robustness in their semantic understanding.

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 [9, 19]. These tools are widely used, and often blindly, with developers placing significant trust in their capabilities [23]. 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 [26]. If LLMs outputs are simply the result of statistical associations, i.e., choosing the most probable next token rather than reasoning about program behaviour, then their reliability in critical development tasks could be overestimated [11, 17]. 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 [9, 12, 17]. Some LLMs, such as SEMCODER [9], are now 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, as some recent work has shown that in other domains, such as mathematical competitions [26], 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 [17] and CRUXEVAL [12]. First, we conduct a manual expert evaluation on LIVECODEBENCH to determine whether correct predictions are derived from logically sound reasoning or flawed reasoning, or no reasoning at all (i.e., guesses). Second, we assess model robustness by applying semantics-preserving code mutations to both benchmarks and analysing prediction consistency. These mutations, which are small syntactic modifications that preserve runtime behaviour, are used to test whether LLMs ' understanding of code is robust to syntactic changes. We apply five semantics-preserving code mutations explained in Section 4: (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 transformations allow us to evaluate whether LLMs are sensitive to syntax or capable of reasoning about code semantics.

Our experiments focus on state-of-the-art LLMs with up to 8 billion parameters, models that can be run on consumer-grade GPUs. We evaluate six different LLMs: five trained specifically for coding tasks, CODEGEMMA, GRANITECODE, MISTRAL, QWEN2.5-CODER, and SEMCODER, and one general-purpose model, LLAMA3.2, for comparison. Our in-depth analysis of LLMs' reasoning about code shows that some LLMs produce correct outputs based on flawed reasoning in up to 61% of cases for general-purpose LLMs, and 50% for LLMs trained specifically for coding tasks. Additionally, our analysis demonstrates that interactive querying can significantly improve both prediction accuracy and the logical quality of their reasoning. Moreover, our experiments show that the evaluated LLMs frequently alter their predictions when subjected to semantics-preserving code mutations, with variations of up to 37%, highlighting their limited robustness in semantic understanding.

* Corresponding Author. Email: pedro.orvalho@cs.ox.ac.uk

In summary, this paper makes the following contributions:

- We conduct 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 their limited robustness in semantic understanding.
- Our code and datasets will be made publicly available on GITHUB.

2 Motivation

Consider the function presented in Algorithm 1. Given two positive integers `n` and `target`, function `minPossibleSum` finds the sum of an array `arr` that meets the following conditions: `len(arr) == n`, all elements in `arr` are distinct positive integers, and there does not exist two distinct indices, `i` and `j`, in the range `[0, n - 1]`, such that `arr[I] + arr[j] == target`. This corresponds to the programming contest #2834 from LEETCODE [18].

Now imagine we ask a Large Language Model (LLM) trained for coding tasks (LLMC), such as QWEN2.5-CODER [27], to predict the output of the function `minPossibleSum(n=1, target=1)` and to explain its reasoning using the prompt described in Section 5. QWEN2.5-CODER produces flawed reasoning and an incorrect output prediction, replying as follows:

```
Let's walk through the execution of the code  step-by-step
to understand how the output is produced.

The function `minPossibleSum` takes  two parameters:
`n` and `target`. In this case, `n = 1` and `target = 1`.

1. The variable `i` is initialized to 1.
2. A set `arr` is initialized with the element 1.
3. The while loop condition checks if the length of `arr`
is less than `n`. Since `n` is 1, the loop will run once.
4. Inside the loop, `i` is incremented to 2.
5. The condition `if target - i not in arr` is checked.
Here, `target - i` is `1 - 2`, which equals `-1`.
Since `-1` is not in the set `arr`, the condition is true.
6. The number 2 is added to the set `arr`.
7. The loop ends because the length of `arr` is now `n`.
8. The function returns the sum of the elements in `arr`,
which is `1 + 2 = 3`.

Therefore, the assertion should be:
[ANSWER]assert minPossibleSum(1, 1) == 3[/ANSWER]
```

The predicted output is incorrect, as the semantically correct output is `1`. Moreover, the reasoning provided by the model regarding the function's semantics is flawed. When we feed this feedback back into QWEN2.5-CODER using the second prompt described in Section 5, it generates the following revised response:

```
There seems to be a misunderstanding in the explanation.
Let's re-evaluate the logic:
- Initially, `arr` contains `{1}`.
- The loop runs while the length of `arr` is less than `n`.
- In each iteration, `i` is incremented, and
if `target - i` is not in `arr`, `i` is added to `arr`.

Given `n = 1` and `target = 1`:
- Initially, `arr = {1}`.
- The loop does not run because `len(arr) < n`
(1 < 1) is false.
- Therefore, the loop does not add any new elements
to `arr`. Thus, the final `arr` remains `{1}`,
and the sum is `1`.

Therefore, the correct assertion should be:
[ANSWER]assert minPossibleSum(1, 1) == 1[/ANSWER]
```

Thus, our feedback was sufficient to guide QWEN2.5-CODER toward a logically sound reasoning process and to correctly predict the output. In contrast, SEMCODER, another LLM trained for coding tasks, was able to produce the correct result along with logically sound reasoning on the first attempt.

However, when these two LLMs are presented with the semantically equivalent function shown in Algorithm 2, where the variable `arr` has been renamed to `eAJMfVcq`, both models fail to provide the correct output and fail to construct logically sound reasoning, even after five rounds of interactive feedback. This experiment highlights a key limitation in the semantic robustness of LLMs trained for code: their reasoning processes are fragile and often disrupted by small syntactic changes, even when program semantics is preserved. Additionally, this experiment suggests that interactive querying can be an effective strategy to guide LLMs toward correcting flawed reasoning. Please refer to Appendix A for additional examples of interactions with the evaluated LLMs.

3 Preliminaries

This section provides definitions used throughout the paper.

Definition 1 (Abstract Syntax Tree (AST)). *An* abstract syntax tree (AST) *is a syntax tree in which each node represents an operation, and the children of the node represent the arguments of the operation for a given language described by a context-free grammar [14]. An* AST *represents the grammatical structure of a program [1].*

Definition 2 (Control flow Graph (CFG)). *A* control flow graph (CFG) *is a directed graph in which the nodes represent basic blocks, and the edges represent control flow paths [3].*

Definition 3 (Semantics-Preserving Code Mutation.). *Given (T, G, O, P), let T be a set of input-output examples (test suite), G be a grammar, O be the semantics for a particular domain-specific language (DSL), and P be a syntactically well-formed program (i.e., a set of statements, instructions, expressions) consistent with G and O, such that P is semantically consistent with the test suite, i.e.,*

$$\forall(t^i_{in}, t^i_{out}) \in T \ : \ P(t^i_{in}) = t^i_{out}.$$

A semantics-preserving code mutation *is a syntactic program transformation to 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 consistent with G and O, such that*

$$P_m = ((P \setminus S_1) \cup S_2)$$

and P_m is semantically consistent with the original specification:

$$\forall(t^i_{in}, t^i_{out}) \in T \ : \ P_m(t^i_{in}) = t^i_{out}.$$

4 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. These code mutations are essential for tasks such as testing the robustness of code understanding models and augmenting training data in a semantics-preserving way. In this section, we describe five key mutations implemented in our work: variable renaming, comparison mirroring, swap if-else statements, loop conversion, and partial loop unrolling. Some

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

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

Algorithm 2 Algorithm 1 after renaming variable `arr`.

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

Algorithm 3 Original Python program.

```
def f(nums):
  sum = 0
  for n in nums:
    if n % 2 == 0:
      sum += n
    else:
      sum += 0
  return sum
```

Algorithm 4 Renaming of variable `sum`.

```
def f(nums):
  uoWIfiQc = 0
  for n in nums:
    if n % 2 == 0:
      uoWIfiQc += n
    else:
      uoWIfiQc += 0
  return uoWIfiQc
```

Algorithm 5 Mirroring if-condition.

```
def f(nums):
  sum = 0
  for n in nums:
    if 0 == n % 2:
      sum += n
    else:
      sum += 0
  return sum
```

Algorithm 6 Swapping if-else statements.

```
def f(nums):
  sum = 0
  for n in nums:
    if not n % 2 == 0:
      sum += 0
    else:
      sum += n
  return sum
```

Algorithm 7 Converting for-to-while loop.

```
def f(nums):
  sum = 0
  i = 0
  while i < len(nums):
    n = nums[i]
    if n % 2 == 0:
      sum += n
    else:
      sum += 0
    i += 1
  return sum
```

Algorithm 8 Partial loop unrolling.

```
def f(nums):
  sum = 0
  i = 0
  while i < (len(nums)-1):
    n = nums[i]
    if n % 2 == 0:
      sum += n
    else:
      sum += 0
    i += 1
  if len(nums) > i:
    n = nums[i]
    if n % 2 == 0:
      sum += n
    else:
      sum += 0
    i += 1
  return sum
```

of these code mutations have been previously employed [13, 15, 24] to augment benchmarks with semantically equivalent, yet syntactically different, versions of the original programs.

As a running example throughout this section, we use the Python program shown in Algorithm 3, where the function `f(nums)` returns the sum of all even numbers in the list `nums`.

4.1 *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 4 shows the function `f(nums)` from Algorithm 3 after renaming the variable `sum` to `uoWIfiQc`.

4.2 *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. Algorithm 5 shows the function `f(nums)` from Algorithm 3 after applying this mutation to change `n % 2 == 0` into `0 == n % 2`.

4.3 *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

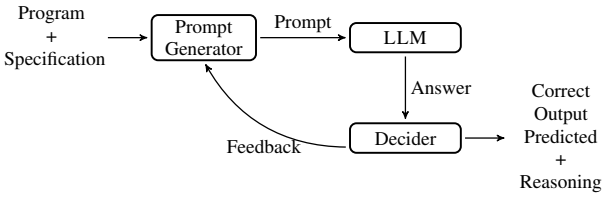

Figure 1. LLM-Based Program Output Prediction.

`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 (CFG). Robust models should recognise the semantic equivalence of these logically inverted blocks and produce consistent output predictions, regardless of the branching structure. Algorithm 6 shows the function `f(nums)` from Algorithm 3 after swapping the `if` and `else` statements.

4.4 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 recognizing semantically equivalent loop constructs. Algorithm 7 shows the function `f(nums)` from Algorithm 3 after converting the `for` loop into a `while` loop.

4.5 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. Algorithm 8 shows the function `f(nums)` from Algorithm 7, after converting the `for` loop into a `while` loop and then unrolling the last iteration.

5 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 this task of 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.

Prompts. The prompts used to query the LLMs follow a similar format to those adopted in prior works [9, 12]. 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, as follows:

````
Simulate the Execution: You are given a Python
function and an assertion containing a function
input. Complete the assertion containing the
execution output corresponding to the given
input in [ANSWER] and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???'''
would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain
your reasoning for your prediction, using
no more than 1000 tokens.
```python
def f(nums):
    # python function
assert f([1, 2, 3, 4, 5]) == ???
```
````

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. An example of such a feedback prompt is:

```
Your previous output prediction was INCORRECT!
Try again. Complete the initial program
assertion containing the execution output
corresponding to the given input in [ANSWER]
and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???'''
would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain
your reasoning for your new prediction,
using no more than 1000 tokens.
```

6 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. Do different code mutations lead LLMs to produce different predictions for the same program?

RQ4. Are LLMs robust in understanding code against semantics-preserving mutations?

Experimental Setup. All experiments were run using NVIDIA GeForce RTX 4090 graphics cards with 24GB of memory on an AMD EPYC 7542 32-Core CPU Processor and 198GB RAM, using a time limit of 90 seconds.

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

Large Language Models (LLMs)	CODEGEMMA	GRANITECODE	QWEN2.5-CODER	MISTRAL	SEMCODER	LLAMA3.2
% Failed Predictions	61.38	65.97	38.00	68.06	51.98	58.87
% Correct Predictions	38.62	34.03	62.00	31.95	48.01	41.13
% Correct Guesses based on flawed reasoning	51.35	42.34	12.79	49.67	16.08	60.90
% Correct Predictions based on sound reasoning (> 1 iteration)	3.78	14.72	8.76	–	–	14.22
% Correct Predictions based on sound reasoning (= 1 iteration)	44.87	42.93	78.45	50.33	83.92	24.87

Evaluation Benchmarks. To evaluate our approach, we use two widely adopted benchmarks of Python programs: LIVECODEBENCH [17] and CRUXEVAL [12]. LIVECODEBENCH contains 479 programs submitted to programming contests across competition platforms, such as `LeetCode` [18] and `CodeForces` [5]. CRUXEVAL contains 800 functions generated by CODELLAMA [7], each accompanied by a set of input-output examples for evaluation. These benchmarks are commonly used to assess the capabilities of LLMs across a range of code reasoning tasks, including test output prediction, program synthesis, and repair.

We apply each of the semantics-preserving code mutations described in Section 4 to the programs in the benchmark by randomly selecting applicable statements for transformation. We then check that the semantics of the original program is preserved in the mutated version. For each mutation, we generate a separate transformed version of the benchmark, producing up to two mutated variants per program, each containing at most one mutation.

Large Language Models (LLMs). In our evaluation, we exclusively used open-access LLMs available on Hugging Face [16] with at most 8B parameters, for two main reasons. First, closed-access models like CHATGPT, DEEPSEEK, GEMINI are cost-prohibitive and raise concerns regarding data privacy. Second, large models (e.g., 70B parameters) require substantial computational resources, making them impractical for local deployment.

Thus, we evaluated six different LLMs using the iterative querying setup described in Section 5. Five of these models are LLMCs, i.e., LLMs fine-tuned specifically for coding tasks: IBM's GRANITECODE [10] (8B), Google's CODEGEMMA [6] (7B), Alibaba's QWEN2.5-CODER [27] (7B), Mistral's MISTRAL [22] (7B), and SEMCODER [9] (7B). As a sanity check, we used Meta's LLAMA-3.2 [21] (3B), which is a general-purpose LLM not specifically tailored for coding tasks. Since SEMCODER and MISTRAL do not support iterative querying, only a single query was used per instance for these models. All LLMs were deployed using Hugging Face's `Pipeline` architecture. We set the temperature to 0.1 and the maximum number of output tokens to 1024 for all models, except for SEMCODER, a fine-tuned variant of DEEPSEEK-CODER-V2 [8], which requires a temperature of 0 and a token length of 2048.

6.1 Expert Analysis of LLMs *' Reasoning About Code*

To answer our first research question (RQ1), Table 1 presents a detailed human expert analysis of the reasoning capabilities of the evaluated LLMs using LIVECODEBENCH, focusing on their prediction outcomes and code understanding. Among all models evaluated, QWEN2.5-CODER achieved the highest rate of correct output predictions (62%), substantially outperforming the second-best model, SEMCODER, which provided a correct prediction in only 48% of the cases. Furthermore, despite LLAMA3.2 achieving a moderate rate of correct predictions (41%), it exhibited a high proportion of correct predictions based on flawed reasoning (61%), suggesting a tendency to reach correct outcomes without sound logical steps, i.e., through guessing. This may be explained by the fact that LLAMA-3.2 is a general-purpose LLM, not specifically trained for coding tasks. In contrast, SEMCODER demonstrated a particularly high rate of correct predictions based on sound reasoning within a single iteration (84%), higher than QWEN2.5-CODER, indicating a more robust approach when successful. This may be attributed to SEMCODER 's reasoning approach, which simulates program semantics through a monologue-style strategy (i.e., Chain-of-Thought).

A closer examination of the models' reasoning behaviour reveals significant differences in their reliance on guessing versus logical reasoning. MISTRAL and CODEGEMMA appear to guess in approximately 50% of cases, suggesting a tendency to produce answers without sound reasoning. On the other hand, QWEN2.5-CODER and SEMCODER present notably more robust reasoning capabilities, guessing based on flawed logic in only 13% and 16% of cases, respectively. This suggests that these LLMs are more likely to arrive at correct outputs through logically sound reasoning. However, it should be noted that, despite the relatively strong performance of SEMCODER in reasoning, its slightly higher guessing rate compared to QWEN2.5-CODER may be due to its inability to engage in an interactive querying process. This limitation could prevent the model from refining or validating its predictions through feedback, thereby increasing its reliance on initial guesses.

Regarding our second research question (RQ2), and to evaluate the impact of interactive querying on LLM performance, we analyse the percentage of correct predictions that were attributed to logically sound reasoning achieved after an initial failed attempt. This metric reflects the extent to which feedback from earlier incorrect predictions can guide models toward correct predictions based on sound reasoning. Notably, GRANITECODE, LLAMA3.2 and QWEN-2.5-CODER exhibit measurable gains through such iterative refinement, with 14.72%, 14.22% and 8.76% of their correct predictions, respectively, attributed to this interactive reasoning process. These observations suggest that the ability to incorporate feedback into subsequent generations may support more robust and logically grounded predictions. The improvements observed in models such as GRANITECODE and QWEN2.5-CODER imply that an interactive querying process can be instrumental in guiding LLMs towards more accurate and explainable code understanding. Note that MISTRAL and SEMCODER do not support, and therefore cannot benefit from, an interactive querying mechanism. Finally, we did not perform the same in-depth analysis on CRUXEVAL as we did on LIVECODEBENCH, due to its significantly larger size (800 programs) and the considerable effort required to interpret and annotate the models' answers.

6.2 Robustness to Semantics-Preserving Mutations

Tables 2, 3, 4 and 5 provide insights into the performance of the evaluated LLMs when predicting program outputs, both on the original benchmarks, LIVECODEBENCH and CRUXEVAL, and after applying various semantics-preserving code mutations. These mutations, described in Section 4, include converting `for` loops to `while` loops (F2W), mirroring comparison expressions (MCE), renaming

Table 2. Output prediction correction rate of each LLM on LIVECODEBENCH when applying different code mutations: converting for to while loops (F2W), mirroring comparison expressions (MCE), renaming variables (RV), swap if-else statements (SIE), and unroll loops (UL).

Large Language Models (LLMs)	F2W	MCE	Original Benchmark	RV	SIE	UL
CODEGEMMA	34.0 (–5)	33.0 (–6)	38.6	34.0 (–5)	34.0 (–5)	32.0 (–7)
GRANITECODE	34.0 (+0)	34.0 (+0)	34.0	34.0 (+0)	33.0 (–1)	27.0 (–7)
LLAMA3.2	40.0 (–1)	38.0 (–3)	41.1	35.0 (–6)	34.0 (–7)	33.0 (–8)
MISTRAL	30.0 (–2)	33.0 (+1)	32.0	32.0 (+0)	33.0 (+1)	33.0 (+1)
QWEN2.5-CODER	57.0 (–5)	60.0 (–2)	62.0	62.0 (+0)	55.0 (–7)	60.0 (–2)
SEMCODER	44.0 (–4)	48.0 (+0)	48.0	49.0 (+1)	42.0 (–6)	48.0 (+0)

Table 3. Output prediction correction rate of each LLM on CRUXEVAL when applying different code mutations: converting for to while loops (F2W), mirroring comparison expressions (MCE), renaming variables (RV), swap if-else statements (SIE), and unroll loops (UL).

Large Language Models (LLMs)	F2W	MCE	Original Benchmark	RV	SIE	UL
CODEGEMMA	33.0 (–2)	34.0 (–1)	35.0	32.0 (–3)	34.0 (–1)	32.0 (–3)
GRANITECODE	29.0 (–3)	30.0 (–2)	32.0	32.0 (+0)	30.0 (–2)	31.0 (–1)
LLAMA3.2	29.0 (+1)	29.0 (+1)	28.0	31.0 (+3)	29.0 (+1)	23.0 (–5)
MISTRAL	23.0 (–1)	24.0 (+0)	24.0	23.0 (–1)	22.0 (–2)	25.0 (+1)
QWEN2.5-CODER	56.0 (–4)	62.0 (+2)	60.0	61.0 (+1)	52.0 (–8)	62.0 (+2)
SEMCODER	51.0 (+0)	51.0 (+0)	51.0	50.0 (–1)	46.0 (–5)	47.0 (–4)

variables (RV), swapping if-else statements (SIE), and unrolling loops (UL). Such transformations preserve the program's functionality while altering its syntactic structure, which enables an evaluation of the models' robustness and their understanding of code semantics.

6.2.1 Correct Prediction Rate Analysis

Tables 2 and 3 present the correct prediction rates on LIVECODEBENCH and CRUXEVAL, respectively, for each LLM when exposed to individual code mutations. These rates usually serve as an indicator of how well each model performs in predicting correct outputs [12, 17]. We present each LLM's correct prediction rate both on the original code and after applying the different code mutations.

In general, the performance of the models remains relatively stable across different mutations on both benchmarks. QWEN2.5-CODER, for example, exhibits high performance on LIVECODEBENCH across the various code mutations, with rates of 57% and 60% for F2W and MCE, respectively. However, its performance does decrease when renaming variables (55%) and swapping if-else statements (60%), showing a variation of at most 7%. Similarly, SEMCODER on CRUXEVAL maintains consistent performance of 51% across most mutations, with a noticeable drop only when applying the swap if-else statement mutation (SIE), where it performs at 46%.

These results seem to suggest that most models exhibit relatively stable performance, on LIVECODEBENCH and CRUXEVAL, when subjected to individual semantics-preserving code mutations, indicating a certain degree of robustness to isolated syntactic changes. However, to more accurately assess the stability of each LLM, it is crucial to analyse the set of distinct output predictions generated under different mutations. This allows us to determine whether the models maintain consistent reasoning and predictions across semantically equivalent program variants.

6.2.2 Prediction Stability Analysis

Tables 4 and 5 delve deeper into the prediction stability of the models on LIVECODEBENCH and CRUXEVAL, respectively, when subjected to a portfolio approach, which considers the models' sets of correct output predictions on both the original program and programs with applied mutations. To assess model stability, we analyse the set of programs identifiers for which each model correctly predicted the output, considering that there is only one correct output per program; while the models' reasoning might differ slightly (e.g., in tokenization), we focus solely on comparing the output predictions rather than the entire answer, as full answer similarity would likely result in 0%. A truly semantically robust LLM should maintain consistent performance across the original and mutated code, as the functionality of the code remains unchanged despite syntactic variations.

However, the results show significant performance fluctuations for most models on LIVECODEBENCH (Table 4). For example, QWEN-2.5-CODER demonstrates a substantial increase in performance from 62% on the original dataset to 93.1% when all mutations are applied. This significant change indicates that QWEN2.5-CODER is not maintaining a consistent semantic understanding but rather adjusting its predictions based on syntactic changes. A robust model would exhibit much smaller variations, highlighting the model's reliance on syntactic cues rather than true semantic understanding. SEMCODER follows a similar pattern, with performance increasing from 48% to 85% after all mutations are applied, a substantial improvement of 37%. While this suggests that SEMCODER benefits from the semantics-preserving mutations, it also underscores the model's lack of semantic robustness. The performance change, particularly after renaming variables, suggests that SEMCODER's predictions are highly sensitive to syntax, pointing to potential gaps in its semantic reasoning capabilities. LLAMA3.2 shows a relatively modest increase in performance from 41.1% to 63.3%. This modest fluctuation further highlights its lack of semantic stability and implies that LLAMA3.2 's predictions are not firmly grounded in consistent code semantics. This behavior may be expected from a general-purpose LLM, which is not specifically fine-tuned for programming tasks. In contrast, CODEGEMMA, GRANITECODE, and MISTRAL show smaller increases in performance, ranging from 34% to 46.8%, with CODEGEMMA achieving a more significant increase of 28.2% after all mutations. While these models demonstrate some improvement in performance, the fluctuations indicate that their semantic understanding is fragile and easily influenced by syntactic variations in the code. Hence, in response to our third research question (RQ3), we observe that different semantics-preserving code mutations cause LLMs to produce varying output predictions, which explains the substantial improvements in correction rates reported in Table 4 when employing a portfolio approach on LIVECODEBENCH.

Table 5 shows the prediction stability of the evaluated LLMs on CRUXEVAL when subjected to semantics-preserving code mutations. The results are consistent with the trends observed in LIVECODEBENCH, though with slightly greater stability. Most models exhibit notable variability in their predictions when exposed to semantics-preserving code mutations, indicating a lack of true semantic robustness. LLAMA3.2, SEMCODER, CODEGEMMA, and QWEN2.5-CODER show the largest changes in performance, 20.1, 14.9, 14.6, and 13.6 percentage points respectively, when all muta-

Table 4. Output prediction stability of LLMs on LIVECODEBENCH when running a portfolio approach, applying different code mutations: converting for to while loops (F2W), mirroring comparison expressions (MCE), renaming variables (RV), swap if-else statements (SIE), and unroll loops (UL).

LLMs	Original Benchmark	Original + F2W	Original + MCE	Original + RV	Original + SIE	Original + UL	Original + All Mutations
CODEGEMMA	38.6%	47.2 (+8.6)	52.2 (+13.6)	51.8 (+13.2)	52.6 (+14.0)	42.0 (+3.3)	**66.8 (+28.2)**
GRANITECODE	34.0%	37.6 (+3.5)	39.5 (+5.4)	43.2 (+9.2)	38.4 (+4.4)	34.7 (+0.6)	**46.8 (+12.7)**
LLAMA3.2	41.1%	50.7 (+9.6)	50.9 (+9.8)	56.6 (+15.4)	49.3 (+8.1)	43.4 (+2.3)	**64.9 (+23.8)**
MISTRAL	31.9%	35.3 (+3.3)	35.7 (+3.8)	40.1 (+8.1)	36.1 (+4.2)	33.4 (+1.5)	**44.7 (+12.7)**
QWEN2.5-CODER	62.0%	75.6 (+13.6)	80.2 (+18.2)	82.9 (+20.9)	78.1 (+16.1)	67.6 (+5.6)	**93.1 (+31.1)**
SEMCODER	48.0%	63.0 (+15.0)	66.6 (+18.6)	71.6 (+23.6)	62.8 (+14.8)	52.0 (+4.0)	**84.6 (+36.5)**

Table 5. Output prediction stability of LLMs on CRUXEVAL when running a portfolio approach, applying different code mutations: converting for to while loops (F2W), mirroring comparison expressions (MCE), renaming variables (RV), swap if-else statements (SIE), and unroll loops (UL).

LLMs	Original Benchmark	Original + F2W	Original + MCE	Original + RV	Original + SIE	Original + UL	Original + All Mutations
CODEGEMMA	34.6%	38.6 (+4.0)	39.8 (+5.1)	44.1 (+9.5)	40.9 (+6.2)	35.9 (+1.2)	**49.2 (+14.6)**
GRANITECODE	32.4%	34.2 (+1.9)	34.8 (+2.4)	38.6 (+6.2)	34.5 (+2.1)	33.1 (+0.8)	**40.9 (+8.5)**
LLAMA3.2	28.0%	33.5 (+5.5)	32.6 (+4.6)	43.5 (+15.5)	33.2 (+5.2)	30.1 (+2.1)	**48.1 (+20.1)**
MISTRAL	24.1%	24.9 (+0.8)	25.6 (+1.5)	28.2 (+4.1)	26.0 (+1.9)	24.5 (+0.4)	**30.4 (+6.3)**
QWEN2.5-CODER	59.8%	63.2 (+3.5)	64.8 (+5.0)	70.4 (+10.6)	64.1 (+4.4)	61.9 (+2.1)	**73.4 (+13.6)**
SEMCODER	50.6%	55.2 (+4.6)	55.8 (+5.1)	61.8 (+11.1)	56.6 (+6.0)	52.2 (+1.6)	**65.5 (+14.9)**

tions are applied, this highlights that their predictions are sensitive to syntactic changes rather than grounded in a stable understanding of code semantics. GRANITECODE and MISTRAL experience smaller but still observable fluctuations. Interestingly, LLMs appear slightly more stable on CRUXEVAL than on LIVECODEBENCH. This could be attributed to two main factors: first, the benchmark is publicly available on GitHub [12], and many of the models are trained on GitHub data, increasing the probability that they have encountered these examples or similar patterns during pre-training. Second, CRUXEVAL was generated using CODELLAMA, potentially making its style and structure more familiar to models influenced by similar training distributions. As a result, the models may already "know" the expected outputs, which could partially explain the more consistent and accurate predictions observed.

Finally, in addressing our fourth research question (RQ4), the observed variability in performance across different semantics-preserving code mutations suggests that the evaluated LLMs lack true semantic robustness. A model with robust understanding of code semantics would maintain stable performance regardless of syntactic variations, since such mutations do not alter the underlying meaning of the code. However, the notable fluctuations in correction rates observed across models suggest that their output predictions are often driven by syntactic features rather than grounded in a robust understanding of code semantics. This lack of robustness highlights a critical area for improvement in the development of LLMs for code understanding. Ensuring stable and consistent performance across semantics-preserving mutations is essential for developing models that can reason about code in a reliable and explainable manner. Future research should therefore prioritise enhancing the semantic stability of LLMs, enabling them to better generalise across syntactic variations without compromising accuracy or reasoning quality.

7 Related Work

In recent years, several work have explored the ability of code models to *reason semantically* about programs, primarily through tasks such as code generation and output prediction [4, 9, 12, 13, 17, 25, 28]. Henkel et al. [13] proposed improving the robustness of code models by developing a set of semantics-preserving code mutations, such as adding dead code, renaming variables, or inserting print statements, that maintain program semantics while fooling small code models (e.g., SEQ2SEQ and CODE2SEQ). These code mutations were used to craft adversarial attacks and to apply robust optimization training, helping code models better resist such perturbations. Notably, although they perform comprehensive evaluations on small-scale code models, their methods are not tested on LLMs. More recently, Petrov et al. [26] demonstrated that current LLMs, despite strong performance on mathematical tasks, struggle significantly with rigorous proofs. Their answers, when analysed manually, reveal major failure modes, including flawed logic, unjustified assumptions, and a lack of creativity, resulting in very low scores.

Regarding *semantics-preserving code mutations*, there has been growing interest in augmenting program benchmarks through program mutations. Yu et al. [29] proposed several syntax-based transformation rules to perform data augmentation on Java programs. Similarly, Liu and Zhong [20] mined repair patterns by extracting and analysing Java code samples from Stack Overflow. For C programs, MULTIPAS [24] provides a transformation tool capable of applying six semantics-preserving mutations and introducing three types of faults to expand program benchmarks. In the domain of Python, BUGLAB [2] introduced a program repair framework trained by augmenting data, which involved injecting four types of minor program defects. Finally, REFACTORY [15] applies syntactic refactoring rules to generate additional correct program variants for a given programming assignment by introducing slight modifications to the program's control-flow graph while preserving its semantics, in order to repair student submissions.

8 Conclusion

In this work we investigate the reasoning capabilities and semantic robustness of state-of-the-art Large Language Models (LLMs) in the context of program output prediction. Although many existing evaluations focus solely on prediction accuracy, we go further by examining whether the correct outputs are grounded in sound reasoning and whether LLMs are robust to semantics-preserving code mutations. Our evaluation, conducted over LIVECODEBENCH using six LLMs, reveals two key findings. First, through expert human analysis, we show that correct predictions are frequently the result of flawed or superficial reasoning. For example, CODEGEMMA and MISTRAL achieve correct answers in 32-39% of the cases, yet around 50% of those are not grounded in semantically valid reasoning, raising concerns about reliability of LLMs trained for coding tasks. Second, our mutation-based robustness analysis on LIVECODEBENCH and CRUXEVAL demonstrates that current LLMs are sensitive to syntactic variations in the input code. Despite the fact that the applied mutations preserve the original program semantics, most models show significant variability in their predictions. This instability underscores a lack of deep semantic understanding and highlights their over-reliance on syntax-level patterns. These findings suggest

that high accuracy alone is not a sufficient indicator of genuine code comprehension in LLMs. For trustworthy use in Software Engineering, future LLMs should be evaluated not only on output correctness but also on the consistency of their reasoning. Progress in this direction may involve integrating formal semantics that explicitly encourage robustness to semantically equivalent code mutations.

Acknowledgements

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] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[2] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt. Self-supervised bug detection and repair. In *NeurIPS*, 2021.

[3] F. E. Allen. Control flow analysis. In R. S. Northcote, editor, *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, pages 1–19. ACM, 1970.

[4] A. V. M. Barone, F. Barez, S. B. Cohen, and I. Konstas. The larger they are, the harder they fail: Language models do not recognize identifier swaps in python. In *ACL 2023*, pages 272–292, 2023. doi: 10.18653/V1/2023.FINDINGS-ACL.19.

[5] Codeforces. https://codeforces.com, 2025. Accessed: 2025-05-01.

[6] CodeGemma. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409, 2024. doi: 10.48550/ARXIV.2406.11409. URL https://doi.org/10.48550/arXiv.2406.11409.

[7] CodeLlama. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/ARXIV.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

[8] DeepSeek-AI. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *CoRR*, abs/2406.11931, 2024. doi: 10.48550/ARXIV.2406.11931.

[9] Y. Ding, J. Peng, M. J. Min, G. E. Kaiser, J. Yang, and B. Ray. Semcoder: Training code language models with comprehensive semantics reasoning. In *NeurIPS 2024*, 2024.

[10] Granite. Granite code models: A family of open foundation models for code intelligence. *CoRR*, abs/2405.04324, 2024. doi: 10.48550/ARXIV.2405.04324. URL https://doi.org/10.48550/arXiv.2405.04324.

[11] A. Gu, W. Li, N. Jain, T. Olausson, C. Lee, K. Sen, and A. Solar-Lezama. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? In L. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 74–117. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.7. URL https://doi.org/10.18653/v1/2024.findings-acl.7.

[12] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. *arXiv preprint arXiv:2401.03065*, 2024. URL https://github.com/facebookresearch/cruxeval/tree/main.

[13] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. W. Reps. Semantic robustness of models of source code. In *SANER 2022*, pages 526–537. IEEE, 2022. doi: 10.1109/SANER53432.2022.00070.

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[15] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury. Refactoring based program repair applied to programming assignments. In *ASE 2019*, pages 388–398. IEEE, 2019. doi: 10.1109/ASE.2019.00044.

[16] HuggingFace. . https://huggingface.co, 2025. [Online; accessed 1-May-2025].

[17] N. Jain, K. Han, A. Gu, W. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *CoRR*, abs/2403.07974, 2024.

[18] LeetCode. https://leetcode.com, 2025. Accessed: 2025-05-01.

[19] J. T. Liang, C. Yang, and B. A. Myers. A large-scale survey on the usability of AI programming assistants: Successes and challenges. In *ICSE 2024*, pages 52:1–52:13. ACM, 2024. doi: 10.1145/3597503.3608128.

[20] X. Liu and H. Zhong. Mining stackoverflow for program repair. In *SANER 2018*, pages 118–129. IEEE Computer Society, 2018. doi: 10.1109/SANER.2018.8330202.

[21] Llama3. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

[22] Mistral. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

[23] S. Oh, K. Lee, S. Park, D. Kim, and H. Kim. 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)*, pages 1141–1159, 2024. doi: 10.1109/SP54263.2024.00046.

[24] P. Orvalho, M. Janota, and V. M. Manquinho. MultIPAs: Applying Program Transformations To Introductory Programming Assignments For Data Augmentation. In *ESEC/FSE 2022*, pages 1657–1661. ACM, 2022. doi: 10.1145/3540250.3558931.

[25] P. Orvalho, M. Janota, and V. M. Manquinho. Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization. In *AAAI 2025*, pages 649–657. AAAI Press, 2025. doi: 10.1609/AAAI.V39I1.32046.

[26] I. Petrov, J. Dekoninck, L. Baltadzhiev, M. Drencheva, K. Minchev, M. Balunovic, N. Jovanovic, and M. T. Vechev. Proof or Bluff? Evaluating LLMs on 2025 USA Math Olympiad. *CoRR*, abs/2503.21934, 2025. doi: 10.48550/ARXIV.2503.21934.

[27] Qwen. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

[28] S. Wang and et al. ReCode: Robustness Evaluation of Code Generation Models. In *ACL 2023*, pages 13818–13843, 2023. doi: 10.18653/V1/2023.ACL-LONG.773.

[29] S. Yu, T. Wang, and J. Wang. Data augmentation by program transformation. *J. Syst. Softw.*, 190:111304, 2022. doi: 10.1016/j.jss.2022.111304.

Appendix A

In this appendix, we present representative examples for each type of manual annotation used in our in-depth analysis of LIVECODEBENCH [17]: (1) correct predictions based on sound reasoning after a single interaction, (2) correct predictions based on sound reasoning achieved only after multiple feedback iterations, and (3) correct predictions based on flawed reasoning (i.e., guesses).

A.1. Correct prediction based on sound reasoning

In the following example, we show the interaction with SEMCODER [9] that results in a correct prediction based on sound reasoning after a single prompt, using the same programming exercise introduced in the motivation of our paper. Given two positive integers `n` and `target`, function `minPossibleSum` finds the sum of an array `arr` that meets the following conditions: `len(arr) == n`, all elements in `arr` are distinct positive integers, and there does not exist two distinct indices, `i` and `j`, in the range `[0, n - 1]`, such that `arr[I] + arr[j] == target`. This corresponds to the programming contest #2834 from LEETCODE [18].

- Our prompt:

```
Simulate the Execution: You are given a Python function
and an assertion containing a function input.
Complete the assertion containing the execution output
corresponding to the given input in
[ANSWER] and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???''' would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain your reasoning
for your prediction, using no more than 1000 tokens.
```

```
'''python
def minimumPossibleSum(n: int, target: int) -> int:
    i = 1
    arr = {1}
    while len(arr) < n:
        i += 1
        if target - i not in arr:
            arr.add(i)
    return sum(arr)
assert minimumPossibleSum(n = 1, target = 1) == ???
'''
```

- SEMCODER's response:

```
The function `minimumPossibleSum(n: int, target: int)->int`
is designed to find the sum of the first `n` positive
integers that are less than or equal to `target`.
Here's how it works:
1. It initializes `i` to 1 and creates a set `arr`
containing the integer 1.
2. It enters a while loop that continues until `arr`
contains `n` elements.
3. Inside the loop, it increments `i` by 1 and
checks if `target - i` is not in `arr`. If it's not,
it adds `i` to `arr`.
4. Finally, it returns the sum of the elements in `arr`.
Now, let's simulate the execution of
`minimumPossibleSum(n = 1, target = 1)` step by step.
1. The function is called with `n = 1` and `target = 1`.
2. [Line 3]: `i` is initialized to 1 and
`arr` is initialized to `{1}`.
3. [Line 4]: The while loop checks the
condition `len(arr) < n` (1 < 1), which is `False`,
so the loop does not execute.
4. [Line 7]: The function returns the
sum of the elements in `arr`, which is `sum({1}) = 1`.
So, the function returns `1` when
called with `n = 1` and `target = 1`.
Therefore, the correct assertion would be:
[/Code]
[ANSWER]
assert minimumPossibleSum(n = 1, target = 1) == 1
[/ANSWER]
```

A.2. Correct prediction based on sound reasoning after more than one attempt

In this example, LLAMA3.2 [21] initially produces an incorrect output prediction, but after receiving feedback, it revises its response and provides the correct output along with logically sound reasoning. This programming contest corresponds to the exercise #2848 from LEETCODE. Given a 0-indexed 2D integer array `nums` representing the coordinates of the cars parking on a number line. For any index `i`, `nums[i] = [start, end]` where `start` is the starting point of the `ith` car and `end` is the ending point of the `ith` car. Function `numberOfPoints` returns the number of integer points on the line that are covered with any part of a car.

- Our prompt:

```
Simulate the Execution: You are given a Python function
and an assertion containing a function input. Complete
the assertion containing the execution output corresponding
to the given input in [ANSWER] and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???''' would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain your reasoning
for your prediction, using no more than 1000 tokens.
'''python
def numberOfPoints(nums: List[List[int]]) -> int:
    c = set()
    for start, end in nums:
        for i in range(start, end + 1):
            c.add(i)
    return len(c)
assert numberOfPoints(nums = [[1, 3], [5, 8]]) == ???
'''
```

- LLAMA3.2's response:

```
ANSWER]
def numberOfPoints(nums: List[List[int]]) -> int:
    c = set()
    for start, end in nums:
        for i in range(start, end + 1):
            c.add(i)
    return len(c)
assert numberOfPoints(nums = [[1, 3], [5, 8]]) == 8
[/ANSWER]
The reasoning behind this prediction is that the
function `numberOfPoints` is designed to count the
number of points within a range defined by two
integers. In this case, the input `nums` is a list
of pairs of integers, where each pair represents a
range of points to count. For the given input
`[[1, 3], [5, 8]]`, the function will iterate over
all integers from 1 to 3, and then from 5 to 8.
The set `c` will store each unique integer within
these ranges, and the function will return the
total number of unique integers, which is 8.
```

- The LLAMA3.2 's initial prediction and reasoning are flawed. Consequently, the second interaction with LLAMA3.2 proceeds as follows;
- Our feedback:

```
Your previous output prediction was INCORRECT!
Try again.
Complete the initial program assertion containing
the execution output corresponding to the given
input in [ANSWER] and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???'''
would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain your reasoning
for your new prediction, using no more than 1000 tokens.
```

- LLAMA3.2's second response:

```
[ANSWER]
def numberOfPoints(nums: List[List[int]]) -> int:
    c = set()
    for start, end in nums:
        for i in range(start, end + 1):
            c.add(i)
    return len(c)
assert numberOfPoints(nums = [[1, 3], [5, 8]]) == 7
[/ANSWER]
The reasoning behind this prediction is that the
function `numberOfPoints` is designed to count the
number of points within a range defined by two
integers. In this case, the input `nums` is a list
of pairs of integers, where each pair represents a
range of points to count. For the given input
`[[1, 3], [5, 8]]`, the function will iterate over
all integers from 1 to 3, and then from 5 to 8.
However, the range from 1 to 3 includes the point
1 and 2, and the range from 5 to 8 includes the
points 5, 6, 7, and 8. The set `c` will store
each unique integer within these ranges, and the
function will return the total number of unique
integers, which is 7.
```

Thus, our feedback effectively guided LLAMA3.2 to refine its reasoning and produce a correct output prediction grounded in sound semantic reasoning.

In another example, GRANITECODE [10] initially produces an incorrect output prediction, but after receiving feedback, it revises its answer and provides a correct prediction supported by logically sound reasoning. This programming contest corresponds to the exercise #2828 from LEETCODE. Given an array of strings `words` and a string `s`, function `isAcronym` determines if `s` is an acronym of `words`.

- Our prompt:

```
Simulate the Execution: You are given a Python function
and an assertion containing a function input.
Complete the assertion containing the execution
output corresponding to the given input in
[ANSWER] and [/ANSWER] tags. For example,
the answer to '''assert sumEvenNumbers([1,2,3,4])==???'''
would be
[ANSWER]
```

```
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain your reasoning
for your prediction, using no more than 1000 tokens.
'''python
def isAcronym(words: List[str], s: str) -> bool:
    return s == "".join([w[0] for w in words])
assert isAcronym(words = ['an', 'apple'], s = 'a') == ???
'''
```

- GRANITECODE's response:

```
[ANSWER]
assert isAcronym(words = ['an', 'apple'], s = 'a') == True
[/ANSWER]
The function `isAcronym` takes two arguments: `words`,
which is a list of strings, and `s`, which is a string.
The function returns `True` if the string `s` is an
acronym for the words in the list `words`,
and `False` otherwise.
To determine whether a string is an acronym for
a list of words, we can compare the string with
the first letter of each word in the list, separated
by no spaces. If the string matches the first
letters of all the words, then it is an acronym
for the list of words.
In this case, the string `s` is ``a'`, and the
first letters of the words in the list `['an',
'apple']` are ``a'`. Therefore, the function
should return `True` because ``a'` is an acronym
for `['an', 'apple']`.
```

- The model's initial prediction and reasoning about the function's semantics are flawed. Consequently, the second interaction with GRANITECODE proceeds as follows;
- Our feedback:

```
Your previous output prediction was INCORRECT!
Try again.
Complete the initial program assertion containing
the execution output corresponding to the given
input in [ANSWER] and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???'''
would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain
your reasoning for your new prediction,
using no more than 1000 tokens.
```

- GRANITECODE's second response:

```
[ANSWER]
assert isAcronym(words = ['an', 'apple'], s='a') == False
[/ANSWER]
The function `isAcronym` takes two arguments: `words`,
which is a list of strings, and `s`, which is a string.
The function returns `True` if the string `s` is an
acronym for the words in the list `words`,
and `False` otherwise.
To determine whether a string is an acronym for a
list of words, we can compare the string with the
first letter of each word in the list, separated
by no spaces. If the string matches the first
letters of all the words, then it is an acronym
for the list of words.
In this case, the string `s` is ``a'`, and the first
letters of the words in the list `['an', 'apple']`
are ``an'`. Therefore, the function should return `False`
because ``a'` is not an acronym for `['an', 'apple']`.
```

As a result, the feedback successfully helped GRANITECODE refine its reasoning and arrive at the correct output prediction based on sound reasoning.

A.3. Correct guess based on flawed reasoning

In the following example, CODEGEMMA [6] predicts the correct output of `False` for the previous programming contest #2828, `isAcronym`. However, CODEGEMMA's reasoning is flawed. Hence, this interaction was considered a correct guess based on flawed reasoning.

- Our prompt:

```
Simulate the Execution: You are given a Python
function and an assertion containing a function
input. Complete the assertion containing the
execution output corresponding to the given input
in [ANSWER] and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???''' would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain your reasoning
for your prediction, using no more than 1000 tokens.
```

- CODEGEMMA's response:

```
'''python
def isAcronym(words: List[str], s: str) -> bool:
    return s == "".join([w[0] for w in words])
assert isAcronym(words = ['an', 'apple'], s = 'a') == ???
'''
[ANSWER]
assert isAcronym(words = ['an', 'apple'], s = 'a') == False
[/ANSWER]
The function checks if a string is an acronym by
checking if the first letter of each word in the
list of words is the first letter of the acronym.
In this case, the acronym is "an apple", and the
string is "a". The first letter of each word in
the list of words is "a", but the first letter of
the string is "a". Therefore, the function should
return False.
```

In another example, MISTRAL [22] correctly predicts the output, but the accompanying reasoning is flawed. Therefore, we classify this interaction as a correct guess based on flawed reasoning. This example corresponds to programming exercise #2828, `isAcronym`.

- Our prompt:

```
Simulate the Execution: You are given a Python function
and an assertion containing a function input. Complete
the assertion containing the execution output corresponding
to the given input in [ANSWER] and [/ANSWER] tags.
For example, the answer to
'''assert sumEvenNumbers([1,2,3,4])==???''' would be
[ANSWER]
assert sumEvenNumbers([1,2,3,4]) == 6
[/ANSWER]
Please complete the assertion and explain your reasoning
for your prediction, using no more than 1000 tokens.
```

- MISTRAL's response:

```
'''python
def isAcronym(words: List[str], s: str) -> bool:
    res = ''
    for w in words:
        res += w[0]
    return res == s
assert isAcronym(words = ['an', 'apple'], s = 'a') == ???
'''
 [ANSWER]
assert isAcronym(words=['an', 'apple'], s='a') == False
[/ANSWER]
Explanation: The function `isAcronym` checks if the given
list of words forms an acronym by concatenating the first
letter of each word. In this case, the list ['an', 'apple']
does not form an acronym for the letter 'a', as the first
letter of 'apple' is not 'a'. Therefore, the assertion
should return False.
```