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

Pedro Orvalho [1], and Marta Kwiatkowska [1]

[1]Department of Computer Science, University of Oxford, Oxford, UK

QAVAS Group Meeting, University of Oxford, UK

Oxford, 15 May 2025

Motivation

- The software engineering community has embraced LLM-based tools, such as GitHub Copilot and ChatGPT, to **streamline code workflows, assist in debugging, and even automate code completion**.

Motivation

- The software engineering community has embraced LLM-based tools, such as GitHub Copilot and ChatGPT, to **streamline code workflows, assist in debugging, and even automate code completion**.
- LLMs **are widely used, and often blindly**, with developers placing significant trust in their capabilities [Oh et al., 2024].

Motivation

However, this growing reliance on LLMs for coding tasks raises a fundamental question:

Motivation

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**?

Motivation

- While recent LLMs can produce syntactically correct code, their **responses might just reflect pattern recognition over code syntax** rather than genuine semantic understanding [Petrov et al., 2025].

Motivation

- While recent LLMs can produce syntactically correct code, their **responses might just reflect pattern recognition over code syntax** rather than genuine semantic understanding [Petrov et al., 2025].

- If LLMs outputs are simply the result of statistical associations, then **their reliability in critical development tasks could be overestimated** [Gu et al., 2024].

Motivation

- While recent LLMs can produce syntactically correct code, their **responses might just reflect pattern recognition over code syntax** rather than genuine semantic understanding [Petrov et al., 2025].
- If LLMs outputs are simply the result of statistical associations, then **their reliability in critical development tasks could be overestimated** [Gu et al., 2024].
- In other domains, such as mathematical competitions [Petrov et al., 2025], LLMs tend to **provide accurate predictions, but based on flawed reasoning**.

Motivation

```python
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)
```

Motivation

```python
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)
```

Given two integers `n` and `target`, function `minPossibleSum` finds the sum of an array `arr` such that:

- `len(arr) == n`;
- all elements in `arr` are distinct positive integers;
- $\nexists_{0}^{n-1}(i,j) : arr[i] + arr[j] == target$;

Motivation

```python
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)
assert minPossibleSum(n=1,target=1)==???
```

LLMs for code (LLMCs)

Now imagine we ask Qwen2.5-Coder, to predict this function's output, and to explain its reasoning.

Qwen2.5-Coder’s Answer

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.

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

Qwen2.5-Coder’s Answer

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]

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

Providing some feedback to Qwen2.5-Coder

```
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.
```

Qwen2.5-Coder’s Answer

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 runs once 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]

Qwen2.5-Coder's Answer

```
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 runs once 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]
```

Our feedback was sufficient to **guide Qwen2.5-Coder toward a logically sound reasoning process and to correctly predict the output**!

LLMs' Robustness

LLMs for code (LLMCs)

- When Qwen2.5-Coder and SemCoder are presented with this semantically equivalent function, **both models fail to provide the correct output**;

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

LLMs' Robustness

LLMs for code (LLMCs)

- When Qwen2.5-Coder and SemCoder are presented with this semantically equivalent function, **both models fail to provide the correct output**;
- Both models also **fail to construct logically sound reasoning**, even after five rounds of interactive feedback.

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

In this work

- Conduct a **manual expert evaluation to assess whether LLMs' code output predictions** are based on logically sound reasoning, flawed reasoning, or mere guesses.

In this work

- Conduct a **manual expert evaluation to assess whether LLMs' code output predictions** are based on logically sound reasoning, flawed reasoning, or mere guesses.

- Evaluate LLMs' **output prediction stability** across five different semantics-preserving code mutations.

Semantics-Preserving Code Mutation

Given a program P that is **syntactically well-formed program**, and it is **semantically consistent with the test suite**, i.e.,

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

Semantics-Preserving Code Mutation

Given a program P that is **syntactically well-formed program**, and it is **semantically consistent with the test suite**, i.e.,

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

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, such that

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

and P_m is **syntactically well-formed** and **semantically consistent with the original specification**:

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

Semantics-Preserving Code Mutations

We introduce **five semantics-preserving code mutations** designed to syntactically modify Python programs without altering their semantics:

- variable renaming;
- comparison expression mirroring;
- swapping if-else statements;
- loop conversion
- partial loop unrolling.

Variable renaming

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

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

Comparison Expression Mirroring

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

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

Swap If-Else Statements

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

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

For-to-While Loop Conversion

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

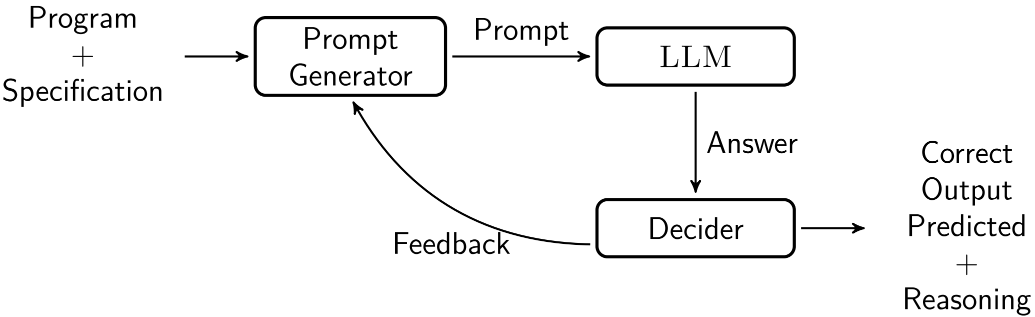
```python
def f(nums):
  sum = 0
  i = 0
  while i < len(nums):
    n = nums[i]
    if n % 2 == 0:
      sum += n
    else:
      sum += 0
    i += 1
  return sum
```

Partial Loop Unrolling

```
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
```

```
def f(nums):
  sum, i = 0, 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
```

LLM-Based Program Output Prediction



Prompt Example

````
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]) == ???
```
~~~

Prompt Example for 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.

```python
def f(nums):
    # python function
assert f([1, 2, 3, 4, 5]) == ???
```
~~~

Experimental Setup

- **Evaluation Benchmarks:**
  - LiveCodeBench [Jain et al., 2024] **contains 479 programs** submitted to programming contests across competition platforms, such as `LeetCode`;

Experimental Setup

- **Evaluation Benchmarks:**
  - LiveCodeBench [Jain et al., 2024] **contains 479 programs** submitted to programming contests across competition platforms, such as `LeetCode`;
  - CruxEval [Gu et al., 2024] **contains 800 functions** generated by CodeLlama, each accompanied by a set of input-output examples for evaluation.

Experimental Setup

- **Evaluation Benchmarks:**
  - LiveCodeBench [Jain et al., 2024] **contains 479 programs** submitted to programming contests across competition platforms, such as `LeetCode`;
  - CruxEval [Gu et al., 2024] **contains 800 functions** generated by CodeLlama, each accompanied by a set of input-output examples for evaluation.
- For each program mutation, **we generate a separate transformed version of the benchmark**, producing up to two mutated variants per program, each containing at most one mutation;

Experimental Setup

- **Evaluation Benchmarks:**
    - LiveCodeBench [Jain et al., 2024] **contains 479 programs** submitted to programming contests across competition platforms, such as `LeetCode`;
    - CruxEval [Gu et al., 2024] **contains 800 functions** generated by CodeLlama, each accompanied by a set of input-output examples for evaluation.
- For each program mutation, **we generate a separate transformed version of the benchmark**, producing up to two mutated variants per program, each containing at most one mutation;
- We also **check that the semantics of the original program is preserved** in the mutated versions.

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **six different LLMs**.

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
  - **Five of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's GraniteCode;
    - Google's CodeGemma;
    - Alibaba's Qwen2.5-Coder;
    - Mistral's Mistral;
    - SemCoder;

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
  - **Five of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's GraniteCode;
    - Google's CodeGemma;
    - Alibaba's Qwen2.5-Coder;
    - Mistral's Mistral;
    - SemCoder;
  - **The other model is a general-purpose LLM**: Meta's Llama3.2;

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
  - **Five of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
    - IBM's GraniteCode;
    - Google's CodeGemma;
    - Alibaba's Qwen2.5-Coder;
    - Mistral's Mistral;
    - SemCoder;
  - **The other model is a general-purpose LLM**: Meta's Llama3.2;
- Experiments were conducted using a timeout of **90s**.

Analysis of LLMs' Reasoning About Code

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

Expert Analysis of LLMs' Reasoning About Code

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 it.)	3.78	14.72	8.76	–	–	14.22
% Correct Predictions based on sound reasoning ($=$ 1 it.)	44.87	42.93	78.45	50.33	83.92	24.87

Expert Analysis of LLMs’ Reasoning About Code

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 it.)	3.78	14.72	8.76	–	–	14.22
% Correct Predictions based on sound reasoning ($=$ 1 it.)	44.87	42.93	78.45	50.33	83.92	24.87

RQ1. Are Large Language Models (LLMs) truly reasoning about code semantics, or merely guessing likely answers?

Expert Analysis of LLMs' Reasoning About Code

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 it.)	3.78	14.72	8.76	–	–	14.22
% Correct Predictions based on sound reasoning ($=$ 1 it.)	44.87	42.93	78.45	50.33	83.92	24.87

RQ2. Does the interactive querying process help LLMs arrive at correct predictions supported by logically sound reasoning?

Robustness to Semantics-Preserving Mutations

LIVECODEBENCH

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 1: 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).

Robustness to Semantics-Preserving Mutations

CruxEval

Large Language Models (LLMs)	**F2W**	**MCE**	**Original Benchmark**	**RV**	**SIE**	**UL**
CodeGemma	33.0 (–2)	34.0 (–1)	35	32.0 (–3)	34.0 (–1)	32.0 (–3)
GraniteCode	29.0 (–3)	30.0 (–2)	32	32.0 (+0)	30.0 (–2)	31.0 (–1)
Llama3.2	29.0 (+1)	29.0 (+1)	28	31.0 (+3)	29.0 (+1)	23.0 (–5)
Mistral	23.0 (–1)	24.0 (+0)	24	23.0 (–1)	22.0 (–2)	25.0 (+1)
Qwen2.5-Coder	56.0 (–4)	62.0 (+2)	60	61.0 (+1)	52.0 (–8)	62.0 (+2)
SemCoder	51.0 (+0)	51.0 (+0)	51	50.0 (–1)	46.0 (–5)	47.0 (–4)

Table 2: 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).

Robustness to Semantics-Preserving Mutations

RQ3. Do different code mutations lead LLMs to produce different predictions for the same program?

Robustness to Semantics-Preserving Mutations

RQ3. Do different code mutations lead LLMs to produce different predictions for the same program?

- **It is crucial to analyse the set of distinct output predictions** generated under different mutations to assess the stability of each LLM.

Robustness to Semantics-Preserving Mutations

RQ3. Do different code mutations lead LLMs to produce different predictions for the same program?

- **It is crucial to analyse the set of distinct output predictions** generated under different mutations to assess the stability of each LLM.
- This allows us to **determine whether the models maintain consistent reasoning and predictions** across semantically equivalent program variants.

Robustness to Semantics-Preserving Mutations

LiveCodeBench

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 3: 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).

Robustness to Semantics-Preserving Mutations

CRUXEVAL

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)**

Table 4: 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).

Robustness to Semantics-Preserving Mutations

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

Robustness to Semantics-Preserving Mutations

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

LLMs	**LiveCodeBench + Mutations**	**CruxEval + Mutations**
CodeGemma	66.8 (+28.2)	49.2 (+14.6)
GraniteCode	46.8 (+12.7)	40.9 (+8.5)
Llama3.2	64.9 (+23.8)	48.1 (+20.1)
Mistral	44.7 (+12.7)	30.4 (+6.3)
Qwen2.5-Coder	93.1 (+31.1)	73.4 (+13.6)
SemCoder	84.6 (+36.5)	65.5 (+14.9)

Takeaway Message

- We investigate the **reasoning capabilities and semantic robustness of Large Language Models (LLMs)** in the context of program output prediction;

Takeaway Message

- We investigate the **reasoning capabilities and semantic robustness of Large Language Models (LLMs)** in the context of program output prediction;
- We examine whether the **correct outputs are grounded in sound reasoning** and whether LLMs are **robust to semantics-preserving code mutations**;

Takeaway Message

- We investigate the **reasoning capabilities and semantic robustness of Large Language Models (LLMs)** in the context of program output prediction;
- We examine whether the **correct outputs are grounded in sound reasoning** and whether LLMs are **robust to semantics-preserving code mutations**;
- Our evaluation using six LLMs, reveals two key findings:

Takeaway Message

- We investigate the **reasoning capabilities and semantic robustness of Large Language Models (LLMs)** in the context of program output prediction;
- We examine whether the **correct outputs are grounded in sound reasoning** and whether LLMs are **robust to semantics-preserving code mutations**;
- Our evaluation using six LLMs, reveals two key findings:
    - Through expert human analysis, we show that **correct predictions are frequently the result of flawed reasoning**.

Takeaway Message

- We investigate the **reasoning capabilities and semantic robustness of Large Language Models (LLMs)** in the context of program output prediction;
- We examine whether the **correct outputs are grounded in sound reasoning** and whether LLMs are **robust to semantics-preserving code mutations**;
- Our evaluation using six LLMs, reveals two key findings:
  - Through expert human analysis, we show that **correct predictions are frequently the result of flawed reasoning**.
    - e.g., CodeGemma and Mistral achieve correct answers in 32-39% of the cases, yet **50% of those are not grounded in valid semantic reasoning**.

Takeaway Message

- We investigate the **reasoning capabilities and semantic robustness of Large Language Models (LLMs)** in the context of program output prediction;
- We examine whether the **correct outputs are grounded in sound reasoning** and whether LLMs are **robust to semantics-preserving code mutations**;
- Our evaluation using six LLMs, reveals two key findings:
  - Through expert human analysis, we show that **correct predictions are frequently the result of flawed reasoning**.
    - e.g., CodeGemma and Mistral achieve correct answers in 32-39% of the cases, yet **50% of those are not grounded in valid semantic reasoning**.
  - LLMs often change predictions in response to our code mutations, indicating **limited robustness in their semantic understanding**.

Thank you!

References

- Oh, Sanghak and Lee, Kiho and Park, Seonhye and Kim, Doowon and Kim, Hyoungshick (2024)
  Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers' Coding Practices with Insecure Suggestions from Poisoned AI Model.
  IEEE Symposium on Security and Privacy, *SP 2024*.
- Ivo Petrov and Jasper Dekoninck and Lyuben Baltadzhiev and Maria Drencheva and Kristian Minchev and Mislav Balunovic and Nikola Jovanovic and Martin T. Vechev (2025)
  Proof or Bluff? Evaluating LLMs on 2025 USA Math Olympiad.
  CoRR 2025.
- Alex Gu and Wen-Ding Li and Naman Jain and Theo Olausson and Celine Lee and Koushik Sen and Armando Solar-Lezama (2024)
  The Counterfeit Conundrum: Can Code Language Models Grasp the Nuances of Their Incorrect Generations?
  ACL 2024.

References

Naman Jain and King Han and Alex Gu and Wen-Ding Li and Fanjia Yan and Tianjun Zhang and Sida Wang and Armando Solar-Lezama and Koushik Sen and Ion Stoica (2024)

LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code.

CoRR 2024.

Alex Gu and Baptiste Rozière and Hugh Leather and Armando Solar-Lezama and Gabriel Synnaeve and Sida I. Wang (2024)

CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution.

CoRR 2024.

CodeLlama Team (2023).

Code Llama: Open Foundation Models for Code.

CoRR 2023.