

From Brittle LLM Code Reasoning to MaxSAT-Based Verified Repairs

Pedro Orvalho

Marie Skłodowska-Curie Postdoctoral Fellow

Institut d'Investigació en Intel·ligència Artificial (IIIA-CSIC), Barcelona, Spain

Previously at:

Department of Computer Science, **University of Oxford**

SSE Seminars @ UCL

London, May 20, 2026



Robustness of LLMs' Code Reasoning

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



Pedro Orvalho, Marta Kwiatkowska. Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations? arXiv:2505.10443 (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

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

Motivation

```
1 def minPossibleSum(n:int, target:int)->int:
2     i = 1
3     arr = {1}
4     while len(arr) < n:
5         i += 1
6         if target - i not in arr:
7             arr.add(i)
8     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;
- $\exists_0^{n-1}(i,j) : arr[i] + arr[j] == target$;

Motivation

```
1 def minPossibleSum(n:int, target:int)->int:
2     i = 1
3     arr = {1}
4     while len(arr) < n:
5         i += 1
6         if target - i not in arr:
7             arr.add(i)
8     return sum(arr)
9 assert minPossibleSum(n=1,target=1)==???
```

LLMs for code (LLMCs)

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.

```
1 def minPossibleSum(n, target):
2     i = 1
3     arr = {1}
4     while len(arr) < n:
5         i += 1
6         if target - i not in arr:
7             arr.add(i)
8     return sum(arr)
9 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]
```

```
1 def minPossibleSum(n, target):
2     i = 1
3     arr = {1}
4     while len(arr) < n:
5         i += 1
6         if target - i not in arr:
7             arr.add(i)
8     return sum(arr)
9 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 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]
```


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 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]
```

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 GPT-5.2 and GEMINI-3 are presented with this semantically equivalent function, **both models fail to provide the correct output;**

```
1 def minPossibleSum(n, target):
2     i = 1
3     eAJMfVcq = {1}
4     while len(eAJMfVcq) < n:
5         i += 1
6         if target - i not in eAJMfVcq:
7             eAJMfVcq.add(i)
8     return sum(eAJMfVcq)
```

LLMs' Robustness

LLMs for code (LLMCs)

- When GPT-5.2 and GEMINI-3 are presented with this semantically equivalent function, **both models fail to provide the correct output**;
- Both models also **fail to construct logically sound reasoning**.

```
1 def minPossibleSum(n, target):
2     i = 1
3     eAJMfVcq = {1}
4     while len(eAJMfVcq) < n:
5         i += 1
6         if target - i not in eAJMfVcq:
7             eAJMfVcq.add(i)
8     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

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

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 ,

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

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

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**

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:

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

```
1 def f(nums):
2     sum = 0
3     for n in nums:
4         if n % 2 == 0:
5             sum += n
6         else:
7             sum += 0
8     return sum
```

```
1 def f(nums):
2     uoWIIfiQc = 0
3     for n in nums:
4         if n % 2 == 0:
5             uoWIIfiQc += n
6         else:
7             uoWIIfiQc += 0
8     return uoWIIfiQc
```

Comparison Expression Mirroring

```
1 def f(nums):
2     sum = 0
3     for n in nums:
4         if n % 2 == 0:
5             sum += n
6         else:
7             sum += 0
8     return sum
```

```
1 def f(nums):
2     sum = 0
3     for n in nums:
4         if 0 == n % 2:
5             sum += n
6         else:
7             sum += 0
8     return sum
```


Swap If-Else Statements

```
1 def f(nums):
2     sum = 0
3     for n in nums:
4         if n % 2 == 0:
5             sum += n
6         else:
7             sum += 0
8     return sum
```

```
1 def f(nums):
2     sum = 0
3     for n in nums:
4         if not n % 2 == 0:
5             sum += 0
6         else:
7             sum += n
8     return sum
```

For-to-While Loop Conversion

```
1 def f(nums):
2     sum = 0
3     for n in nums:
4         if n % 2 == 0:
5             sum += n
6         else:
7             sum += 0
8     return sum
```

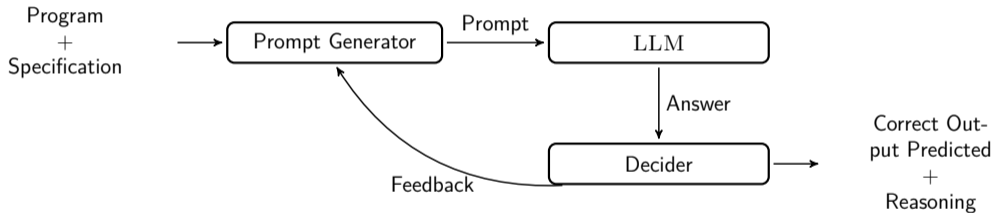
```
1 def f(nums):
2     sum = 0
3     i = 0
4     while i < len(nums):
5         n = nums[i]
6         if n % 2 == 0:
7             sum += n
8         else:
9             sum += 0
10        i += 1
11    return sum
```

Partial Loop Unrolling

```
1 def f(nums):
2     sum = 0
3     i = 0
4     while i < len(nums):
5         n = nums[i]
6         if n % 2 == 0:
7             sum += n
8         else:
9             sum += 0
10        i += 1
11    return sum
```

```
1 def f(nums):
2     sum, i = 0, 0
3     while i < (len(nums)-1):
4         n = nums[i]
5         if n % 2 == 0:
6             sum += n
7         else:
8             sum += 0
9         i += 1
10    if len(nums) > i:
11        n = nums[i]
12        if n % 2 == 0:
13            sum += n
14        else:
15            sum += 0
16        i += 1
17    return sum
```

LLM-Based Program Output Prediction



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**, each program 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**, each program 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 **eight different LLMs**.

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **eight different LLMs**.
 - **Seven of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
 - IBM's GRANITECODE;
 - Google's CODEGEMMA;
 - Mistral's MISTRAL;
 - SEMCODER;
 - Alibaba's QWEN3-CODER;
 - OpenAI's GPT-5.2;
 - Google's GEMINI-3;

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **eight different LLMs**.
 - **Seven of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
 - IBM's GRANITECODE;
 - Google's CODEGEMMA;
 - Mistral's MISTRAL;
 - SEMCODER;
 - Alibaba's QWEN3-CODER;
 - OpenAI's GPT-5.2;
 - Google's GEMINI-3;
 - **The other model is a general-purpose LLM:** Meta's LLAMA3;

Experimental Setup

- **Large Language Models (LLMs):** We evaluated **eight different LLMs**.
 - **Seven of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
 - IBM's GRANITECODE;
 - Google's CODEGEMMA;
 - Mistral's MISTRAL;
 - SEMCODER;
 - Alibaba's QWEN3-CODER;
 - OpenAI's GPT-5.2;
 - Google's GEMINI-3;
 - **The other model is a general-purpose LLM:** Meta's LLAMA3;
- Experiments were conducted using a timeout of **90s**.

Analysis of LLMs' Reasoning About Code

LLMs	CodeGemma	GraniteCode	Qwen3-Coder	Mistral	SemCoder	Llama3	GPT-5.2	Gemini-3
%Failed Predictions	66.8	65.1	49.5	67.4	52.0	61.4	2.5	0.0
%Correct Predictions	33.2	34.9	50.5	32.6	48.0	38.6	97.5	100.0

Expert Analysis of LLMs' Reasoning About Code

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

Expert Analysis of LLMs' Reasoning About Code

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

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

Expert Analysis of LLMs' Reasoning About Code

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

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

Robustness to Semantics-Preserving Mutations

LIVECODEBENCH

LLMs	Original Code	Loop Conversion	Expression Mirroring	Variable Renaming	Swap If-Else	Loop Unrolling
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	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)
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 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

LLMs	Original Code	Loop Conversion	Expression Mirroring	Variable Renaming	Swap If-Else	Loop Unrolling
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	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)
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 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	33.2%	5.8%	8.4%	10.2%	6.5%	1.9%	18.0%
GRANITECODE	34.9%	3.5%	4.0%	6.3%	4.2%	0.4%	10.4%
LLAMA3	38.6%	12.3%	11.3%	16.5%	10.0%	2.5%	26.9%
MISTRAL	32.6%	3.1%	3.3%	7.5%	3.8%	1.7%	11.5%
QWEN3-CODER	50.5%	7.5%	7.5%	10.2%	6.3%	2.3%	14.8%
SEMCODER	48.0%	15.0%	18.6%	23.6%	14.8%	4.0%	36.5%
GPT-5.2	97.1%	1.0%	0.8%	2.1%	1.3%	0.4%	2.7%
GEMINI-3	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

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	30.9%	2.6%	2.8%	8.1%	3.5%	1.0%	11.4%
GRANITECODE	32.6%	1.6%	1.5%	5.9%	2.2%	0.6%	7.9%
LLAMA3	26.5%	4.9%	5.0%	16.0%	6.8%	1.6%	21.4%
MISTRAL	23.8%	0.9%	1.5%	4.5%	2.2%	0.2%	6.6%
QWEN3-CODER	48.9%	1.5%	1.6%	5.0%	2.5%	0.6%	6.4%
SEMCODER	50.6%	4.6%	5.1%	11.1%	6.0%	1.6%	14.9%
GPT-5.2	86.0%	1.9%	1.5%	3.6%	2.2%	0.8%	4.6%
GEMINI-3	76.8%	6.8%	7.4%	14.4%	1.8%	3.4%	15.5%

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	51.2 (+18.0)	42.3 (+11.4)
GRANITECODE	45.3 (+10.4)	40.5 (+7.9)
LLAMA3	65.5 (+26.9)	47.9 (+21.4)
MISTRAL	44.1 (+11.5)	30.4 (+6.6)
QWEN3-CODER	65.3 (+14.8)	55.3 (+6.4)
SEMCODER	84.5 (+36.5)	65.5 (+14.9)
GPT-5.2	99.8 (+2.7)	90.6 (+4.6)
GEMINI-3	100.0 (+0.0)	92.3 (+15.5)

Verifying LLM-based Code with Formal Methods



P. Orvalho, M. Janota, and V. Manquinho. Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault localisation. AAAI 2025.

Motivation

1: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2      int f,s,t;
3      scanf("%d%d%d",&f,&s,&t);
4      if (f < s && f >= t)
5          printf("%d",f);
6      else if (s > f && s >= t)
7          printf("%d",s);
8      else if (t < f && t < s)
9          printf("%d",t);
10
11     return 0;
12 }
```

Motivation

2: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2      int f,s,t;
3      scanf("%d%d%d",&f,&s,&t);
4      if (f < s && f >= t)
5          printf("%d",f);
6      else if (s > f && s >= t)
7          printf("%d",s);
8      else if (t < f && t < s)
9          printf("%d",t);
10
11     return 0;
12 }
```

- Symbolic approaches, while precise, demand an **excessive amount of time to produce an answer**;

Motivation

3: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2      int f,s,t;
3      scanf("%d%d%d",&f,&s,&t);
4      if (f < s && f >= t)
5          printf("%d",f);
6      else if (s > f && s >= t)
7          printf("%d",s);
8      else if (t < f && t < s)
9          printf("%d",t);
10
11     return 0;
12 }
```

- Symbolic approaches, while precise, demand an **excessive amount of time to produce an answer**;
- LLMs, while fast, **often produce incorrect or non-minimal fixes**.

Motivation

4: Semantically incorrect program. Faults: {4,8}.

```
1  int main(){ //finds max of 3 nums
2      int f,s,t;
3      scanf("%d%d%d",&f,&s,&t);
4      if (f < s && f >= t)
5          printf("%d",f);
6      else if (s > f && s >= t)
7          printf("%d",s);
8      else if (t < f && t < s)
9          printf("%d",t);
10
11     return 0;
12 }
```

- Symbolic approaches, while precise, demand an **excessive amount of time to produce an answer**;
- LLMs, while fast, **often produce incorrect or non-minimal fixes**.
- *Idea*: Combine the strengths of **Formal Methods (FM) and LLM-based approaches**.

Program Sketches

5: Semantically incorrect program. Faults :{4,8}.

```
1  int main(){ //finds max of 3 nums
2      int f,s,t;
3      scanf("%d%d%d",&f,&s,&t);
4      if (f < s && f >= t)
5          printf("%d",f);
6      else if (s > f && s >= t)
7          printf("%d",s);
8      else if (t < f && t < s)
9          printf("%d",t);
10
11     return 0;
12 }
```

6: Program sketch with holes.

```
1  int main(){
2      int f,s,t;
3      scanf("%d%d%d",&f,&s,&t);
4      @ HOLE 1 @
5          printf("%d",f);
6      else if (s > f && s >= t)
7          printf("%d",s);
8      @ HOLE 2 @
9          printf("%d",t);
10
11     return 0;
12 }
```

Our Work

- APR problem using an **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach** [Solar-Lezama et al., 2006].

Our Work

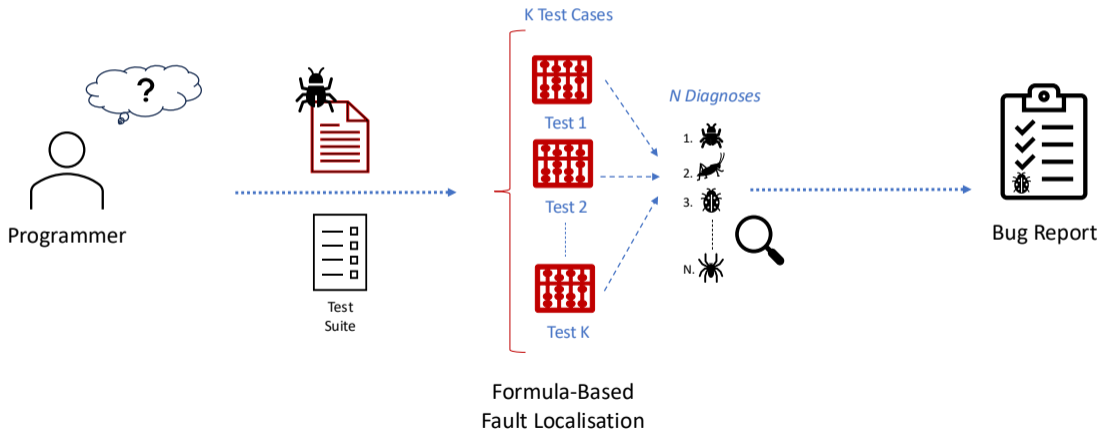
- APR problem using an **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach** [Solar-Lezama et al., 2006].
- Combines the strengths of **Formal Methods (FM)** and **LLM-based approaches**;

Our Work

- APR problem using an **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach** [Solar-Lezama et al., 2006].
- Combines the strengths of **Formal Methods (FM)** and **LLM-based approaches**;
- Uses **MaxSAT-based fault localisation** to rigorously identify buggy lines, which can then be highlighted in the LLM prompt;

Formula-Based Fault Localisation (FBFL)

- FBFL methods encode the localisation problem into **several optimisation problems** to identify a minimal set of bugs (diagnoses).



Formula-Based Fault Localisation (FBFL)

- We formulate the FL problem as a **single optimisation problem**;

P. Orvalho, M. Janota, and V. Manquinho (2024). CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases. *Formal Methods (FM) 2024/*

Formula-Based Fault Localisation (FBFL)

- We formulate the FL problem as a **single optimisation problem**;
- We leverage Maximum Satisfiability (MaxSAT) and the theory of *Model-Based Diagnosis (MBD)* [Reiter et al., 1987], **integrating all failing test cases simultaneously**;

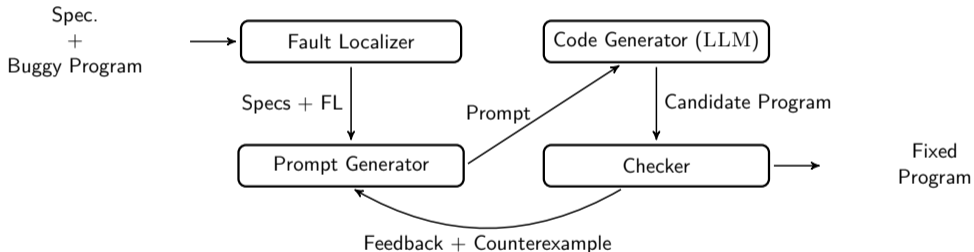
P. Orvalho, M. Janota, and V. Manquinho (2024). CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases. *Formal Methods (FM) 2024/*

Formula-Based Fault Localisation (FBFL)

- We formulate the FL problem as a **single optimisation problem**;
- We leverage Maximum Satisfiability (MaxSAT) and the theory of *Model-Based Diagnosis (MBD)* [Reiter et al., 1987], **integrating all failing test cases simultaneously**;
- We implement this MBD approach in a publicly available tool called CFAULTS [Orvalho et al., 2024].

P. Orvalho, M. Janota, and V. Manquinho (2024). CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases. *Formal Methods (FM) 2024/*

Counterexample Guided Automated Repair



Experimental Setup (2024)

- **Evaluation Benchmark:** C-PACK-IPAS, a set of twenty-five **introductory programming assignments** (IPAS), comprising **1431 faulty programs**;

Experimental Setup (2024)

- **Evaluation Benchmark:** C-PACK-IPAS, a set of twenty-five **introductory programming assignments** (IPAS), comprising **1431 faulty programs**;
- **Large Language Models (LLMs):** We evaluated **six different LLMs**.
 - **Three of these models are LLMCs**, i.e., LLMs fine-tuned for coding tasks:
 - IBM's GRANITECODE;
 - Google's CODEGEMMA;
 - Meta's CODELLAMA.
 - **The other three models are general-purpose LLMs:**
 - Google's GEMMA;
 - Meta's LLAMA3;
 - Microsoft's PHI3.

LLM-Driven APR with CFaults

LLMs	De-TS	De-TS-CE	FIXME_De-TS	FIXME_De-TS-CE	Sk_De-TS	Sk_De-TS-CE	Portfolio (All Configurations)
CodeGemma	597 (41.7%)	606 (42.3%)	592 (41.4%)	601 (42.0%)	682 (47.7%)	688 (48.1%)	823 (57.5%)
CodeLlama	492 (34.4%)	500 (34.9%)	481 (33.6%)	463 (32.4%)	573 (40.0%)	561 (39.2%)	712 (49.8%)
Gemma	496 (34.7%)	492 (34.4%)	446 (31.2%)	444 (31.0%)	532 (37.2%)	534 (37.3%)	670 (46.8%)
Granite	626 (43.7%)	624 (43.6%)	566 (39.6%)	583 (40.7%)	691 (48.3%)	681 (47.6%)	846 (59.1%)
Llama3	564 (39.4%)	590 (41.2%)	535 (37.4%)	557 (38.9%)	578 (40.4%)	591 (41.3%)	851 (59.5%)
Phi3	494 (34.5%)	489 (34.2%)	460 (32.1%)	474 (33.1%)	547 (38.2%)	535 (37.4%)	621 (43.4%)
Portfolio (All LLMs)	842 (58.8%)	846 (59.1%)	796 (55.6%)	820 (57.3%)	900 (62.9%)	907 (63.4%)	1013 (70.8%)

Table 5: The number of programs fixed by each LLM under various configurations. Mapping abbreviations to configuration names: **De** - IPA *Description*, **TS** - *Test Suite*, **CE** - *Counterexample*, **FIXME** - *FIXME Comments*, **SK** - *Sketches*.

LLM-Driven APR with CFaults

LLMs	De-TS	De-TS-CE	FIXME_De-TS	FIXME_De-TS-CE	Sk_De-TS	Sk_De-TS-CE	Portfolio (All Configurations)
CodeGemma	597 (41.7%)	606 (42.3%)	592 (41.4%)	601 (42.0%)	682 (47.7%)	688 (48.1%)	823 (57.5%)
CodeLlama	492 (34.4%)	500 (34.9%)	481 (33.6%)	463 (32.4%)	573 (40.0%)	561 (39.2%)	712 (49.8%)
Gemma	496 (34.7%)	492 (34.4%)	446 (31.2%)	444 (31.0%)	532 (37.2%)	534 (37.3%)	670 (46.8%)
Granite	626 (43.7%)	624 (43.6%)	566 (39.6%)	583 (40.7%)	691 (48.3%)	681 (47.6%)	846 (59.1%)
Llama3	564 (39.4%)	590 (41.2%)	535 (37.4%)	557 (38.9%)	578 (40.4%)	591 (41.3%)	851 (59.5%)
Phi3	494 (34.5%)	489 (34.2%)	460 (32.1%)	474 (33.1%)	547 (38.2%)	535 (37.4%)	621 (43.4%)
Portfolio (All LLMs)	842 (58.8%)	846 (59.1%)	796 (55.6%)	820 (57.3%)	900 (62.9%)	907 (63.4%)	1013 (70.8%)

Table 5: The number of programs fixed by each LLM under various configurations. Mapping abbreviations to configuration names: **De** - IPA *Description*, **TS** - *Test Suite*, **CE** - *Counterexample*, **FIXME** - *FIXME Comments*, **SK** - *Sketches*.

Incorporating **MaxSAT-based Sketches** allows LLMs to repair more programs

On Verifying Python via LLM-Based Transpilation and Bounded Model Checking for C



Pedro Orvalho, Marta Kwiatkowska. On Verifying Python via LLM-Based Transpilation and Bounded Model Checking for C. arXiv:2508.08171 (2025)

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 eight 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 eight 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 eight 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 eight 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**.

Takeaway Message

- We tackle the APR problem using an **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach** [Solar-Lezama et al., 2006];

Takeaway Message

- We tackle the APR problem using an **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach** [Solar-Lezama et al., 2006];
- We employ **MaxSAT-based Fault Localization to guide and minimize LLMs' patches** to incorrect programs by feeding them bug-free program sketches;

Takeaway Message

- We tackle the APR problem using an **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach** [Solar-Lezama et al., 2006];
- We employ **MaxSAT-based Fault Localization to guide and minimize LLMs' patches** to incorrect programs by feeding them bug-free program sketches;
- With our approach **all evaluated LLMs fix more programs and produce smaller patches** than other configurations and symbolic tools;

Takeaway Message

- We tackle the APR problem using an **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach** [Solar-Lezama et al., 2006];
- We employ **MaxSAT-based Fault Localization to guide and minimize LLMs' patches** to incorrect programs by feeding them bug-free program sketches;
- With our approach **all evaluated LLMs fix more programs and produce smaller patches** than other configurations and symbolic tools;
- We also propose **verification and fault localisation for Python programs** by leveraging LLM-based transpilation to C. We are able to accurately **verify small yet non-trivial Python programs where native verification tools fall short.**

Thank you!



This work was supported by HORIZON-MSCA-2025-PF, project 101269051 — Sherlock4Py funded by REA, EU.

References



Oh, Sanghak and Lee, Kiho and Park, Seonhye and Kim, Doowon and Kim, Hyounghick (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.

References



Armando Solar-Lezama and Liviu Tancau and Rastislav Bodík and Sanjit A. Seshia and Vijay A. Saraswat (2018)

Combinatorial sketching for finite programs.

ASPLOS 2006.



Reiter, Raymond (1987)

A Theory of Diagnosis from First Principles.

Artif. Intell. 1987.



P. Orvalho and M. Janota and V. Manquinho (2024)

C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments.

Automated Program Repair (APR) 2024.

References



Edmund M. Clarke and Daniel Kroening and Flavio Lerda (2004)

A Tool for Checking ANSI-C Programs

TACAS 2004.



P. Orvalho and M. Janota and V. Manquinho (2025)

Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault localisation.

AAAI 2025.



P. Orvalho and M. Janota and V. Manquinho (2024)

CFaults: Model-Based Diagnosis for Fault localisation in C with Multiple Test Cases.

Formal Methods (FM) 2024.