# Invitation to Participate in Research Study: Inefficiencies in LLM-generated code

We are conducting a research study titled ***Inefficiencies in LLM-Generated Code*** under the ******. This study is led by ****** (MSc Student), ****** (Postdoctoral Researcher), and ****** (Research Associate), under the supervision of Professor ******.

We invite you to participate in a short survey designed to gather your feedback on identified inefficiencies in LLM-generated code. We will guide you through various categories and subcategories of inefficiencies commonly found in LLM-generated code. For each category, we will provide a description, an example code snippet illustrating the inefficiency, and ask you for your feedback on its relevance and whether you have encountered similar issues.

Moreover, you could kindly share this survey with students/colleagues/friends who you consider eligible to participate. The results of this survey will be publicly accessible through arXiv.org in **anonymized** form.

We greatly appreciate your help in doing so. The survey should take no longer than 10 minutes. At no point in the survey will we ask you for your name, and we will not be logging your IP address to allow anonymity.

For additional details, feel free to contact us at: ******.

* Indicates required question

1. Email

   We will solely use your email to share the results with you.

2. What programming language do you use most frequently? *

*Check all that apply.*

- [ ] Python
- [ ] Java
- [ ] JavaScript
- [ ] C/C++
- [ ] C#
- [ ] Rust
- [ ] GO
- [ ] Other: _____

3. Have you used LLM models to assist you writing code? *

*Mark only one oval.*

- ◯ Yes
- ◯ No

4. What LLM models do you usually use ?

*Check all that apply.*

- [ ] Copilot
- [ ] ChatGPT
- [ ] Gemini
- [ ] Mistral
- [ ] Other: _____

5. What issues have you encountered in the LLM generated code?

6. What is your current job title? For students, please indicate your degree: Master's, PhD, etc. *

7. What is your coding experience? *

*Mark only one oval.*

- 0-1 year
- 1-3 years
- 3-5 years
- More than 5 years

## General Logic

We have identified *General Logic* category which groups inefficiencies related to the logic implemented by the LLM model.

**Wrong Logic:** occurs when the generated code is totally wrong or when it lacks any meaningful logic. E.g. hard-coded values are returned.

*In the example below, the implemented code always returns a constant value, regardless of the input.*

```
def parse_music(music_string: str) -> List[int]:
    """ Input to this function is a string representing musical notes in a special ASCII
format.
    Your task is to parse this string and return list of integers corresponding to how many
beats does each
    not last.

    Here is a legend:
    'o' - whole note, lasts four beats
    'o|' - half note, lasts two beats
    '.|' - quarter note, lasts one beat

    >>> parse_music('o o| .| o| o| .| .| .| .| o o')
    [4, 2, 1, 2, 2, 1, 1, 1, 1, 4, 4]
    """
    return [4, 2, 1, 2, 2, 1, 1, 1, 1, 4, 4]
```

8. How often do you encounter Wrong Logic inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

9. To Which extent *Wrong Logic* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

10. Do you have any additional comment, feedback or suggestion regarding *Wrong Logic*?

**Partially Wong Logic**: occurs when the implementation is not fully correct, e.g some of requirements are implemented with certain parts missing or contain errors.

*In the example below, the implemented code is incomplete. Although the docstring states that sentences are delimited by ('.', '?', '!'), only '.' is handled, other delimiters are ignored.*

```
def is_bored(S):
    """
    You'll be given a string of words, and your task is to count the number
    of boredoms. A boredom is a sentence that starts with the word "I".
    Sentences are delimited by '.', '?' or '!'.

    For example:
    >>> is_bored("Hello world")
    0
    >>> is_bored("The sky is blue. The sun is shining. I love this weather")
    1
    """
    count = 0
    for i in S.split('.'):
        if i.startswith('I'):
            count += 1
    return count
```

11. How often do you encounter Partially Wrong Logic inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

12. To Which extent *Partially Wrong Logic* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

13. Do you have any additional comment, feedback or suggestion regarding *Partially Wrong Logic*?

**Wong Method Input**: occurs when the LLM generates a statement that is generally coherent with the logic but uses incorrect or inappropriate inputs.

*In the example below, the implemented code uses ASCII order (ord()) instead of alphabetical order as intended.*

```
def split_words(txt):
    '''
    Given a string of words, return a list of words split on whitespace, if no whitespaces
exists in the text you should split on commas ',' if no commas exists you should return the
number of lower-case letters with odd order in the alphabet, ord('a') = 0, ord('b') = 1,
ord('z') = 25
    Examples
    split_words("Hello world!") → ["Hello", "world!"]
    split_words("Hello,world!") → ["Hello", "world!"]
    split_words("abcdef") == 3
    '''
    if ' ' in txt:
        return txt.split(' ')
    elif ',' in txt:
        return txt.split(',')
    else:
        return sum(1 for i in range(ord('a'), ord('z')+1) if txt.lower().count(chr(i))%2)
```

14. How often do you encounter Wrong Method Input inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

15. To Which extent *Wrong Method Input* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

16. Do you have any additional comment, feedback or suggestion regarding *Wrong Method Input*?

**General Logic: Summary**

In the General Logic category we found three main sub-categories:

- Wrong Logic: occurs when the entire implemented logic is wrong.
- Partially Wrong Logic: occurs when only some aspects of the requirements are correctly implemented.
- Wrong Method Input: occurs when a statement is generally coherent with the logic but uses incorrect or inappropriate inputs.

17. To Which extent *General Logic* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

18. Have you encountered other inefficiencies in the *General Logic* category that you believe have not been mentioned?

## Performance

We have identified *Performance* category grouping inefficiencies that limit the effectiveness of the generated code such as execution time and memory usage.

**Sub-Optimal Solution (Memory)**: occurs when a more optimal implementation exists, in terms of memory, for the same logic.

In the example below, list comprehension is used, which creates and stores the entire list, including all intermediate values, in memory. This becomes inefficient, especially for large datasets.

```
def create_large_list(n):
      """ Generates a list containing integers from 0 to n-1.
      """
      large_list = [i for i in range(n)]
      return large_list
```

19. How often do you encounter Sub-Optimal Solution (Memory) inefficiencies in LLM-generated code? *

All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

20. To Which extent *Sub-Optimal Solution (Memory)* is relevant to be considered? *

All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

21. Do you have any additional comment, feedback or suggestion regarding *Sub-Optimal Solution (Memory)*?

**Sub-Optimal Solution (Time)**: occurs when a more efficient solution, in terms of execution time, exists for implementing the same logic. Time-related inefficiencies can be categorized based on their underlying causes:

- Sub-Optimal Time Complexity Solution
- Inefficient Iterative Block
- Redundant Steps
- Unnecessary Steps

For this sub-category, we will explore each identified cause in detail, explain it through a code example, and then gather your input regarding its prevalence and importance to address, based on your experience.

Sub-Optimal Solution (Time)

**Sub-Optimal Time Complexity Solution:** occurs when there is a more optimal solution, in terms of Time Complexity.

*In the example below, the n-th Fibonacci number is calculated using a recursive approach with a time complexity of $O(2^n)$. An alternative implementation using memoization, however, reduces the time complexity to O(n).*

```
def fib(n: int):
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

22. How often do you encounter Sub-Optimal Time Complexity Solution inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

23. To Which extent *Sub-Optimal Time Complexity Solution* is relevant to be considered? *

All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

24. Do you have any additional comment, feedback or suggestion regarding *Sub-Optimal Time Complexity Solution*?

Sub-Optimal Solution (Time)

**Unnecessary Steps:** refer to instruction in the code that do not contribute to the final output. Removing these steps has no impact on the result, thereby improving the overall efficiency of the code.

*For example, in the code below, converting a set to a list is unnecessary, as the sorted function can be directly applied to a set. Eliminating the list conversion has no impact on the final output.*

```
def common(l1: list, l2: list):
    """Return sorted unique common elements for two lists.
    >>> common([1, 4, 3, 34, 653, 2, 5], [5, 7, 1, 5, 9, 653, 121])
    [1, 5, 653]
    >>> common([5, 3, 2, 8], [3, 2])
    [2, 3]

    """
    return sorted(list(set(l1) & set(l2)))
```

25. How often do you encounter Unnecessary Steps inefficiencies in LLM-generated code? *

All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

26. To Which extent *Unnecessary Steps* is relevant to be considered? *

All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

27. Do you have any additional comment, feedback or suggestion regarding *Unnecessary Steps*?

Sub-Optimal Solution (Time)

**Redundant Steps**: occurs when the same instruction is executed multiple times.

*In the example below, len(xs) is called at each iteration in the loop, it could be computed once and stored, allowing it to be used at each steps of the loop.*

```
def derivative(xs: list):
    """ xs represent coefficients of a polynomial.
    xs[0] + xs[1] * x + xs[2] * x^2 + ....
     Return derivative of this polynomial in the same form.
    >>> derivative([3, 1, 2, 4, 5])
    [1, 4, 12, 20]
    >>> derivative([1, 2, 3])
    [2, 6]
    """
    return [x * (len(xs) - i) for i, x in enumerate(xs[1:])]
```

28. How often do you encounter Redundant Steps inefficiencies in LLM-generated code? *

All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

29. To Which extent *Redundant Steps* is relevant to be considered? *

All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

30. Do you have any additional comment, feedback or suggestion regarding Redundant Steps?

Sub-Optimal Solution (Time)

**Inefficient Repetitive Block** occurs when the initialization or stopping criteria are overly broad and can be narrowed without affecting the method's output.

*In the example below, the task is to find the largest divisor. Instead of starting the loop from n-1, it can begin from √n, since the largest divisor cannot exceed this value.*

```
def largest_divisor(n: int) -> int:
    """ For a given number n, find the largest number that divides n evenly, smaller
than n
    >>> largest_divisor(15)
    5
    """
    for i in range(n-1, 0, -1):
        if n % i == 0:
            return i
```

31. How often do you encounter Inefficient Repetitive Block inefficiencies in LLM-generated code? *

All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

32. To Which extent *Inefficient Repetitive Block* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

33. Do you have any additional comment, feedback or suggestion regarding *Inefficient Repetitive Block*?

**Performance: Summary**

The Performance category includes two main sub-categories:

- Sub-Optimal Solution (Time): When a more efficient solution in terms of execution time is possible nay be caused by: Sub-Optimal Time Complexity Solution, Inefficient Iterative Block, Redundant Steps or Unnecessary Steps
- A More Optimal Solution Exists (Memory): When a more efficient solution in terms of memory usage is available.

34. To Which extent *Performance* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

35. Have you encountered other inefficiencies in the Performance category that you believe have not been mentioned?

## Readability

This category covers inefficiencies related to the Readability of the code.

**Confusing Naming** occurs when the names of variables or methods are ambiguous, leading to potential misunderstandings. This includes cases where a common method name is used or when a predefined name is redefined.

*In the example below, a variable named sum is used, which can be easily confused with the built-in sum() function.*

```
def double_the_difference(lst):
    '''
    Given a list of numbers, return the sum of squares of the numbers
    in the list that are odd. Ignore numbers that are negative or not
integers.
    '''
    # Your code here
    sum = 0
    for i in lst:
        if i > 0 and i % 2 == 1:
            sum += i * i
    return sum
```

36. How often do you encounter Confusing Naming inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

37. To which extent *Confusing Naming* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

38. Do you have any additional comment, feedback or suggestion regarding *Confusing Naming*?

**Sub-Readable Code Exists:** This occurs when there is a simpler and clearer way to write code regardless of correctness.

*In the example below, the combination of reduce, lambda, and list comprehension makes the code difficult to read and time-consuming to understand.*

```
def digits(n):
    """Given a positive integer n, return the product of the odd digits.
    Return 0 if all digits are even.
    For example:
    digits(1)  == 1
    digits(4)  == 0
    digits(235) == 15
    """
    if n == 0:
        return 0
    else:
        return reduce(lambda x, y: x * y, [int(i) for i in str(n) if int(i) % 2 == 1])
```

39. How often do you encounter Sub-Readable Code Exists inefficiencies in LLM-generated code? *

All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

40. To Which extent *Sub-Readable Code* Exists is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

41. Do you have any additional comment, feedback or suggestion regarding *Sub-Readable Code Exists*?

**Readability: Summary**

The Readability category includes two main sub-categories:

- Confusing Naming: occurs when the names of variables or methods are ambiguous, leading to potential misunderstandings
- Sub-Readable Code Exists: occurs when there is a simpler and clearer way to write code regardless of correctness.

42. To which extent *Readability* is relevant to be considered? *

All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

43. Have you encountered other inefficiencies in the Readability category that you believe have not been mentioned?

_____

## Maintainability

This category encompasses inefficiencies that result in closed code, which limits maintainability.

**Code Duplication**: This occurs when identical or very similar code blocks are repeated in different places within a code, often due to copy-pasting.

*In the example below, the two methods, encode_cyclic() and decode_cyclic(), contain exactly the same implementation.*

```
def encode_cyclic(s: str):
    """
    returns encoded string by cycling groups of three characters.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless the group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    return "".join(groups)


def decode_cyclic(s: str):
    """
    takes as input string encoded with encode_cyclic function. Returns decoded string.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless the group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    return "".join(groups)
```

44. How often do you encounter Code Duplication inefficiencies in LLM-generated code?

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

45. To which extent *Code Duplicatio*n is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

46. Do you have any additional comment, feedback or suggestion regarding *Code Duplication?*

**Comment Duplication:** occurs when one or more comment are duplicated within the code.

```
def largest_prime_factor(n: int):
    """Return the largest prime factor of n. Assume n > 1 and is not a prime.
    >>> largest_prime_factor(13195)
    29
    >>> largest_prime_factor(2048)
    2
    """
    # TODO: Implement this function
    # Hint: Do not use any built-in functions to find prime numbers
    # Hint: Do not use any built-in functions to find prime factors
    # Hint: Do not use any built-in functions to find the largest prime factor
    # Hint: Do not use any built-in functions to find the largest prime factor
    # Hint: Do not use any built-in functions to find the largest prime factor
    # Hint: Do not use any built-in functions to find the largest prime factor
    # Hint: Do not use any built-in functions to find the largest prime factor
```

47. How often do you encounter Comment Duplication inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

48. To which extent *Comment Duplication* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

49. Do you have any additional comment, feedback or suggestion regarding *Comment Duplication*?

**Inefficient Conditional Block:** occurs when a conditional block can be removed or simplified. We categorize Inefficient Conditional Blocks into two types:

- Unnecessary Conditional Block
- Unnecessary Else

For this sub-category, we will explore each type in detail, explain it through a code example, and then gather your input regarding its prevalence and importance to address, based on your experience.

**Unnecessary Conditional Block** occurs when a conditional block is not needed and can be eliminated.

*In the example below, the conditional block can be replaced by returning the condition itself.*

```
def is_positive(x):
    """
    Check if the given number is positive.
    """
    if x > 0:
        return True
    return False
```

50. How often do you encounter Unnecessary Conditional Block inefficiencies in LLM-generated code? *

All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

51. To which extent *Unnecessary Conditional Block* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

52. Do you have any additional comment, feedback or suggestion regarding *Unnecessary Conditional Block*?

**Unnecessary Else Statement:** occurs when removing the else statement does not affect the output.

*In the example below, the else statement is unnecessary because it follows a return, allowing it to be omitted without impacting the result.*

```
from typing import List, Tuple

def sum_product(numbers: List[int]) -> Tuple[int, int]:
        """ For a given list of integers, return a tuple consisting of a sum and a product
of all the integers in a list.
        Empty sum should be equal to 0 and empty product should be equal to 1.
        >>> sum_product([])
        (0, 1)
        >>> sum_product([1, 2, 3, 4])
        (10, 24)
        """
        if len(numbers) == 0:
                return 0, 1
        else:
                return sum(numbers), reduce(lambda x, y: x * y, numbers)
```

53. How often do you encounter Unnecessary Else inefficiencies in LLM-generated code? *

All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

54. To which extent *Unnecessary Else* is relevant to be considered? *

All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

55. Do you have any additional comment, feedback or suggestion regarding *Unnecessary Else Statement*?

**Summary: Maintainability**

The Maintainability category includes three main sub-categories:

- Code Duplication: occurs when identical or very similar code blocks are repeated in different places within a code.
- Comment Duplication: occurs when one or more comment are duplicated within the code.
- Inefficient Conditional Block: occurs when a conditional block is not needed, can be removed or simplified.

56. To which extent *Maintainability* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

57. Have you encountered other inefficiencies in the Maintainability category that you believe have not been mentioned?

_____

Errors

This category encompasses inefficiencies related to fundamental correctness of the code regardless of the logic being implemented.

**Missing Import:** This error occurs when a library or module is used in the code without being imported.

*In the example below, the re module is utilized in the code below without an import statement.*

```
def fix_spaces(text):
    """
    Given a string text, replace all spaces in it with underscores,
    and if a string has more than 2 consecutive spaces,
    then replace all consecutive spaces with -
    fix_spaces(" Example 2") == "_Example_2"
    fix_spaces(" Example   3") == "_Example-3"
    """
    text = text.replace(" ", "_")
    text = re.sub(r"_+", "-", text)
    return text
```

58. How often do you encounter Missing Import inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

59. To which extent *Missing Import* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

60. Do you have any additional comment, feedback or suggestion regarding *Missing Import*?

**Missing Variable Declaration**: occurs when an undefined variable is used.

*In the example below, the variable planets is utilized without being passed as an argument, declared, or initialized.*

```
def bf(planet1, planet2):
    '''
    There are eight planets in our solar system: Mercury, Venus, Earth,
    Mars, Jupiter, Saturn, Uranus, and Neptune. Write a function that takes the
    names of two planets and returns the names of the planets in between.
    bf("Mercury", "Jupiter") ==> ("Venus", "Earth", "Mars")
    '''
    if planet1 not in planets or planet2 not in planets:
        return ()
    else:
        return tuple(sorted(planets[planet1:planet2]))
```

61. How often do you encounter Missing Variable Declaration inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

62. To which extent *Missing Variable Declaration* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

63. Do you have any additional comment, feedback or suggestion regarding *Missing Variable Declaration*?

**Syntax Error:** This error occurs when the syntax of the code is incorrect, preventing it from being compiled.

*In the example below, the use of ++ is invalid syntax in Python.*

```
def how_many_times(string: str, substring: str) -> int:
    """ Find how many times a given substring can be found in the original
string. Count overlapping cases.
    """
    count = 0
    index = 0
    while index!= -1:
        index = string.find(substring, index)
        if index!= -1:
            count++
            index++
    return count
```

64. How often do you encounter Syntax Error inefficiencies in LLM-generated code? *

*All scores are given on a scale from 1 to 5, where 1 indicates a rare occurrence and 5 indicates a frequent occurrence.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Rare | ◯ | ◯ | ◯ | ◯ | ◯ | Frequent |

65. To which extent *Syntax Error* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

66. Do you have any additional comment, feedback or suggestion regarding *Syntax Error*?

**Summary: Error**

The Error category includes three main sub-categories:

- Missing Import: occurs when a library or module is used in the code without being imported.
- Missing Variable Declaration: occurs when an undefined variable is used.
- Syntax Error: occurs when the syntax of the code is incorrect, preventing it from being compiled.

67. To which extent *Error* is relevant to be considered? *

*All scores are given on a scale from 1 to 5, with 1 indicates low importance and 5 indicates high importance.*

*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Low | ◯ | ◯ | ◯ | ◯ | ◯ | High importance |

68. Have you encountered other inefficiencies in the Error category that you believe have not been mentioned?

Thank you!

Your contribution is valuable and greatly appreciated!

69. Do you believe you encountered any other inefficiencies that were not mentioned? If yes, please provide them in the text below.