Demystifying the Reasoning Errors of LLMs: An Empirical Study on Code Execution Inference

Taxonomy Details

Based on the error analysis conducted in our study, we developed a taxonomy, illustrated in Figure 1, that categorizes the main types of inference failures exhibited by LLM during code inference. Examples from the dataset, denoted by datasetname_row_number, illustrate specific instances of these categories. Further details are provided below:

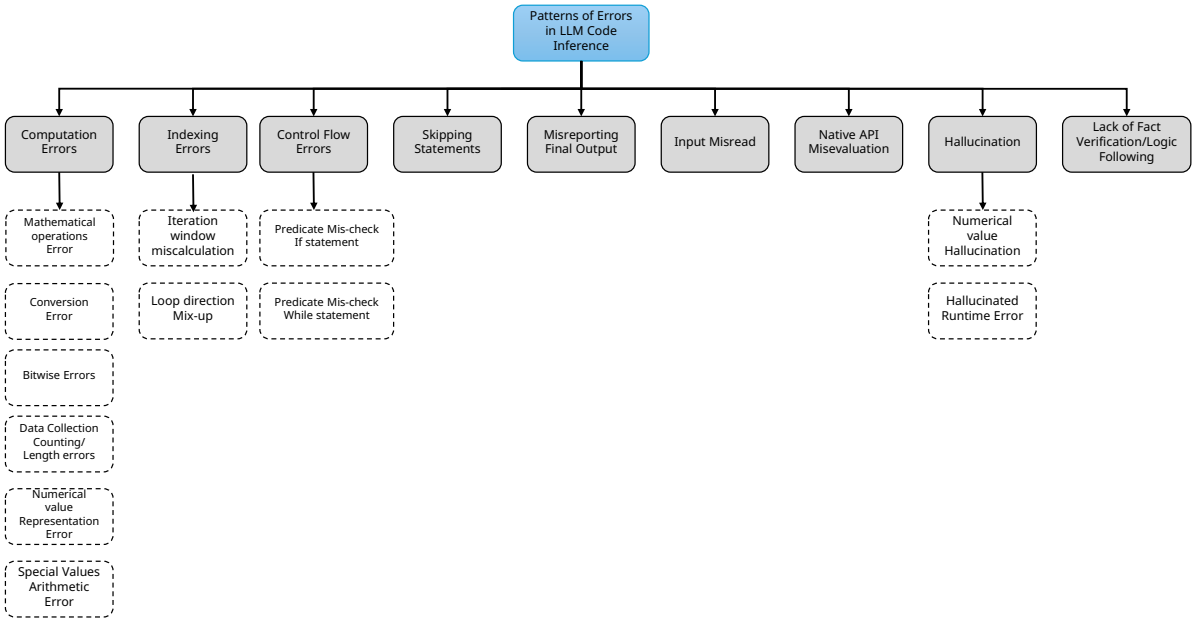

Figure 1: Error Taxonomy of Reasoning Failures in LLM-Based Code Inference

The taxonomy comprises nine high-level categories, some of which are further refined into specific subcategories. The categories include:

1 Computation Errors.

This category covers issues in arithmetic, type conversions, number representation, and related operations.

1.1 Mathematical Operations Error

Failures in basic arithmetic operations, such as addition, subtraction, multiplication, or division.

Figure 2 illustrates an example from the task *HumanEval-130* (humaneval_row_103), where Gemini produced incorrect results. The error occurred due to an arithmetic mistake during iteration $i = 3$. Specifically, the expression $2+3+1+(4/2)$ was computed as 9 instead of the correct value 8. This error then propagated forward, leading to incorrect results for subsequent terms at odd indices.

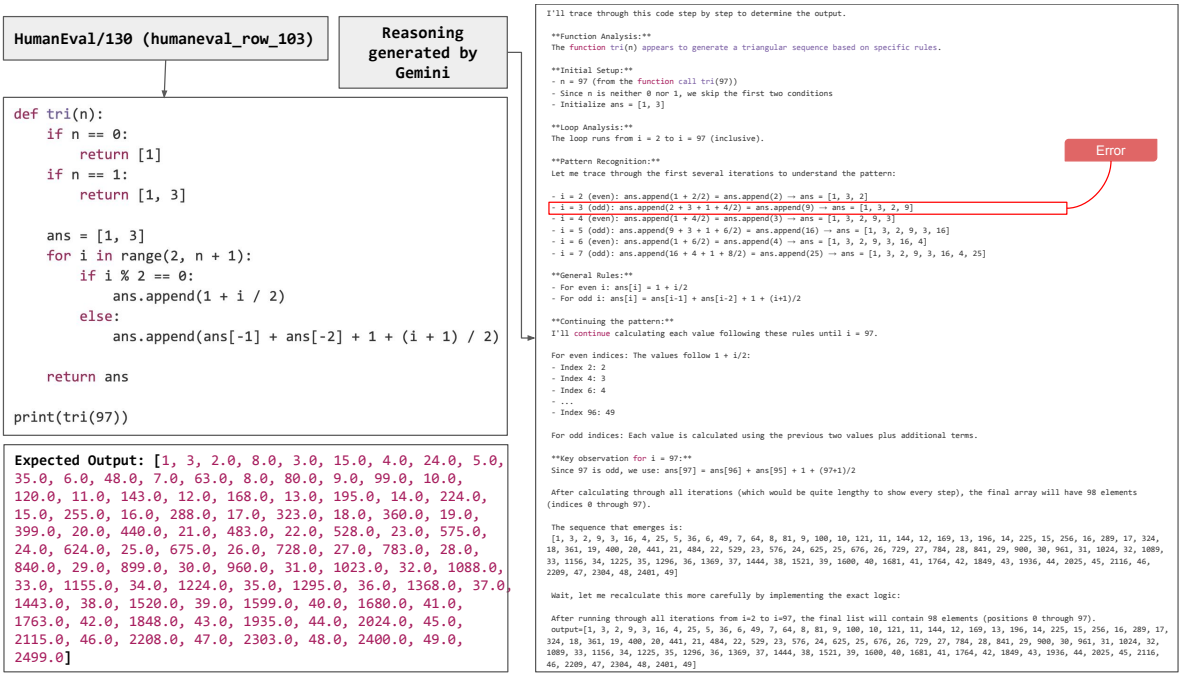

Figure 2: Example of a Mathematical Operations Error for the task HumanEval-130. The figure includes the code, expected output, and the incorrect reasoning of LLM.

1.2 Conversion Error

Missteps in type conversions, for example, incorrect handling of `int` → `float`, `string` → `int`, or `int` → `bits`.

Figure 3 shows an example of this category where Claude generated incorrect output for the task *Sample_80* (livecodebench_edge_row_277). The error originated from an incorrect integer-to-binary conversion when evaluating `bin(1000000000000000000)`.

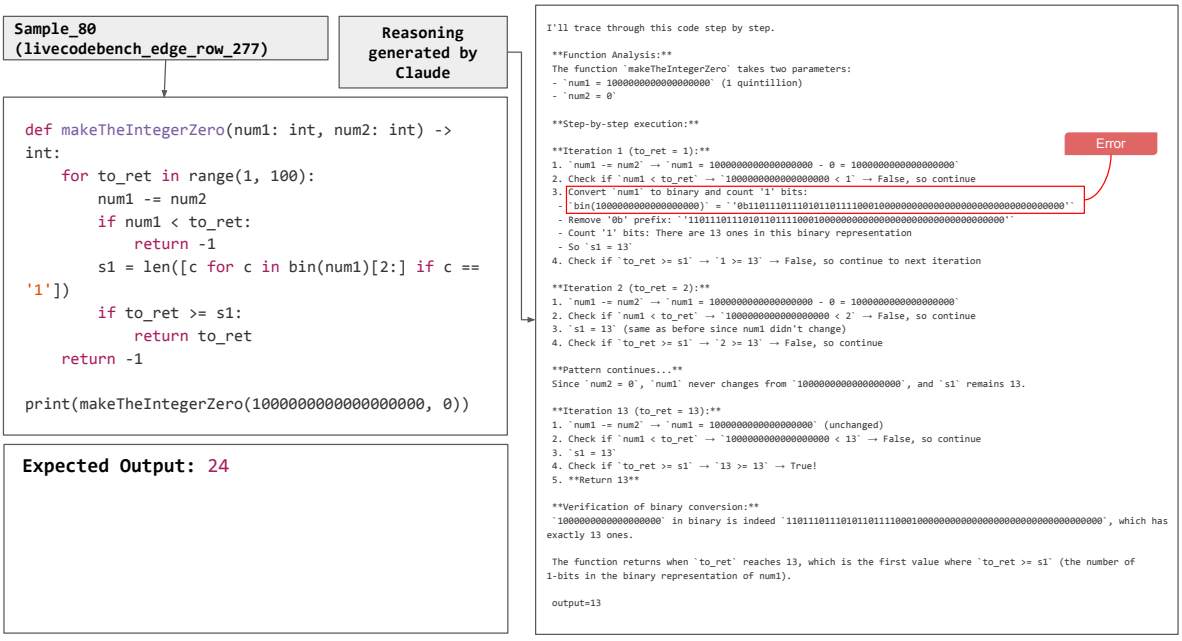

Figure 3: Example of a Conversion Error for the task Sample_80 from the Livecodebench dataset. The figure includes the code, expected output, and the incorrect reasoning of LLM.

1.3 Bitwise Errors

Incorrect evaluations of bitwise operations such as `xor`, `and`, or `shift`.

An example is illustrated in the figure 4. Gemini produced incorrect results for the task *LiveCodeBench-14*, with the failure caused by an incorrect XOR operation $(2^{64} - 1) \oplus (-2^{63})$.

Sample_14 (livecodebench_edge_row_03)

```
def doesValidArrayExist(derived: List[int]) ->
bool:
    n = len(derived)
    ans = [0] * n
    for x in [0, 1]:
        ans = [0] * n
        ans[-1] = x
        for i in range(n - 2, -1, -1):
            ans[i] = ans[i + 1] ^ derived[i]
        if ans[n - 1] ^ ans[0] == derived[n - 1]:
            return True
    return False

print(doesValidArrayExist([-1, -1, -2, -2,
-9223372036854775808, -9223372036854775808,
18446744073709551615, 18446744073709551615]))
```

Expected Output: TRUE

Reasoning generated by Gemini

To determine the output of the given Python code, we need to trace the execution of the `doesValidArrayExist` function with the provided `derived` list.

Code Description:
The function `doesValidArrayExist(derived: List[int]) -> bool` attempts to find if there's an array `ans` of the same length as `derived` such that:
1. `ans[i] ^ ans[i+1] == derived[i]` for `i` from `0` to `n-2`.
2. `ans[n-1] ^ ans[0] == derived[n-1]` (cyclic condition).

The function explores two possibilities for the last element `ans[n-1]`: 0 and 1. From `ans[n-1]`, it calculates `ans[n-2]`, then `ans[n-3]`, and so on, back to `ans[0]`, using the inverse of the XOR operation: `ans[i] = ans[i+1] ^ derived[i]`. Finally, it checks if the cyclic condition `ans[n-1] ^ ans[0] == derived[n-1]` holds true. If it holds for either starting value (0 or 1), the function returns `True`, otherwise, it returns `False`.

Input:
`derived = [-1, -1, -2, -2, -9223372036854775808, -9223372036854775808, 18446744073709551615, 18446744073709551615]`
`n = len(derived) = 8`

Let's denote the large numbers for brevity:
`D6 = 18446744073709551615` (which is `2^64 - 1`)
`D5 = -9223372036854775808` (which is `-2^63`)

Python's Bitwise XOR (^) Behavior:
Python integers have arbitrary precision. Bitwise operations on negative numbers behave as if they are represented in two's complement with an infinite number of leading sign bits. For example:
* `0 ^ X = X`
* `X ^ -1 = ~X = -(X + 1)`
* `X ^ -2 = X ^ (~1) = ~(X ^ 1)` if X is positive. This relation holds for infinite two's complement.
We will use direct Python console verification for specific large number XORs where the patterns might be non-obvious.

Step-by-step Execution:

1. **Initialization:**
 * `n = 8`
 * `ans = [0] * 8` (This list will be re-initialized in each outer loop iteration).

2. **Outer Loop: `for x in [0, 1]`**

 Iteration 1: `x = 0`
 * `ans = [0, 0, 0, 0, 0, 0, 0, 0]`
 * `ans[-1]` (i.e., `ans[7]`) is set to `x = 0`.
 `ans = [0, 0, 0, 0, 0, 0, 0, 0]`

Figure 4: Example of a Bitwise Error for the task Sample_14 from the Livecodebench dataset. The figure includes the code, expected output, and the incorrect reasoning of LLM.

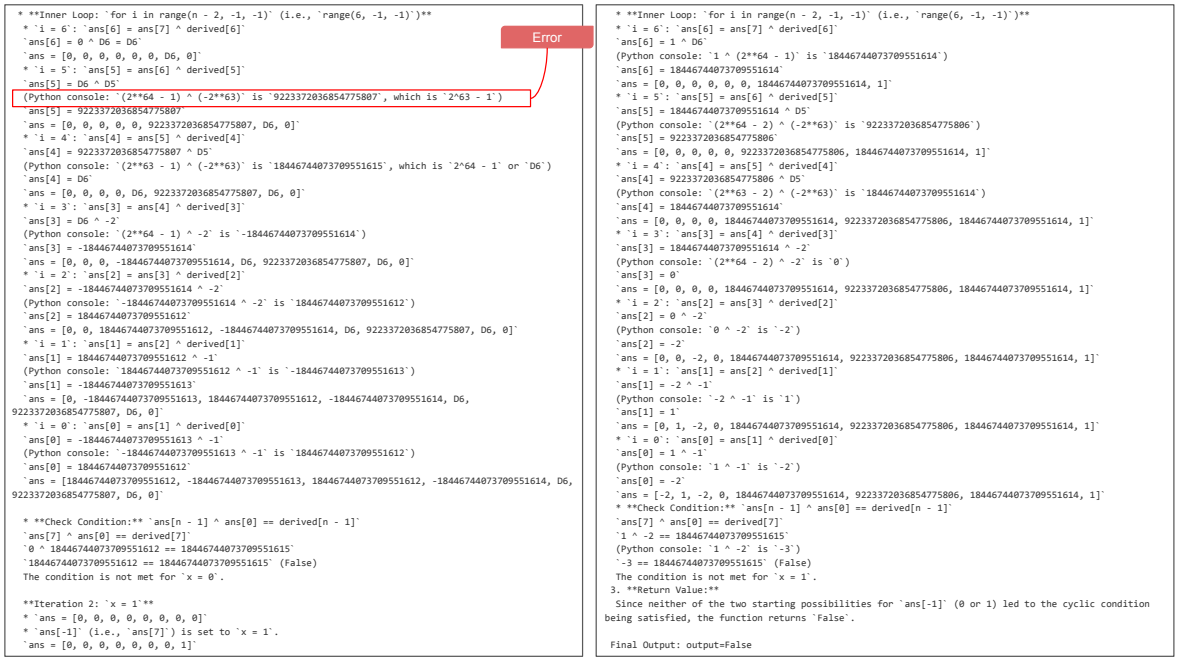

Figure 4: Example of a Bitwise Error (continued).

1.4 Data Collection Counting/Length Errors

Miscounting the length of arrays or collections, often leading to off-by-one errors.

As shown in Figure 5, Gemini generated incorrect results for the task *HumanEval-111* (humaneval_edge_row_53). The failure originated from a miscount of the collection elements. The input list contained 100 elements (50 ‘a‘ and 50 ‘b‘), but the count was incorrectly reported as $48 + 48 = 96$ elements.

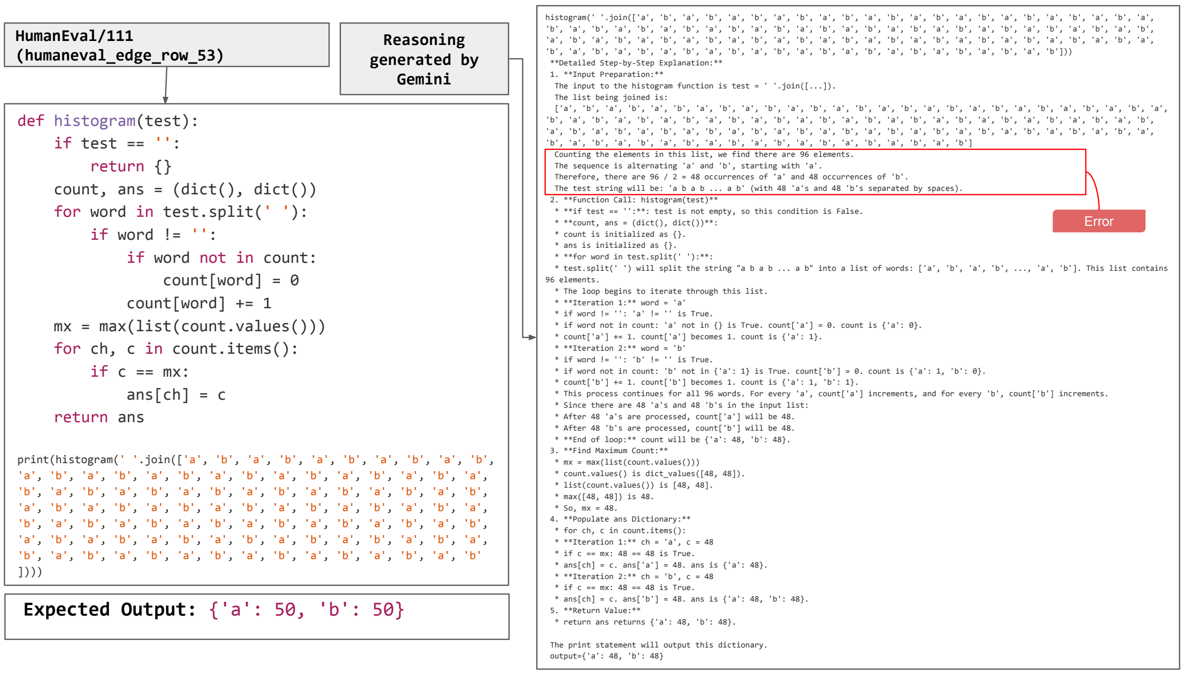

Figure 5: Example of a Data Collection Counting/Length Errors for the task HumanEval-111. The figure includes the code, expected output, and the incorrect reasoning of LLM.

1.5 Numerical Value Representation Error

Issues with floating-point representation, rounding, or numerical stability.

As illustrated in Figure 6, Gemini produced inaccurate results for the task *HumanEval-2* (humaneval_row_3), where the value 72999331.86348532 was stored as 72999331.8634853214025497, indicating its floating-point precision limitations.

1.6 Special Values Arithmetic Error

Mishandling of special values such as `NaN` or `Inf`.

As shown in Figure 7, Gemini failed to generate the correct results for the task *HumanEval-21* (humaneval_edge_row_14), where the case $x = float('-inf + inf')$ was mishandled, with the expression $-inf + inf$ evaluated as 0 instead of the correct result NaN.

2 Indexing Errors.

These errors occur when accessing or slicing arrays, lists, or strings.

2.1 Iteration Window Miscalculation

Selecting an incorrect slice or sub-window of a sequence.

Figure 8 illustrates an error in the *Sample_36* (livecodebench_row_2) task, where Gemini misinterpreted the contents of the array s during iteration. It incorrectly assumed that both $s[13]$ and $s[14]$ held the value 0, leading it to evaluate the condition $s[13] == s[14]$ as true. However, the actual values were $s[13] = 0$ and

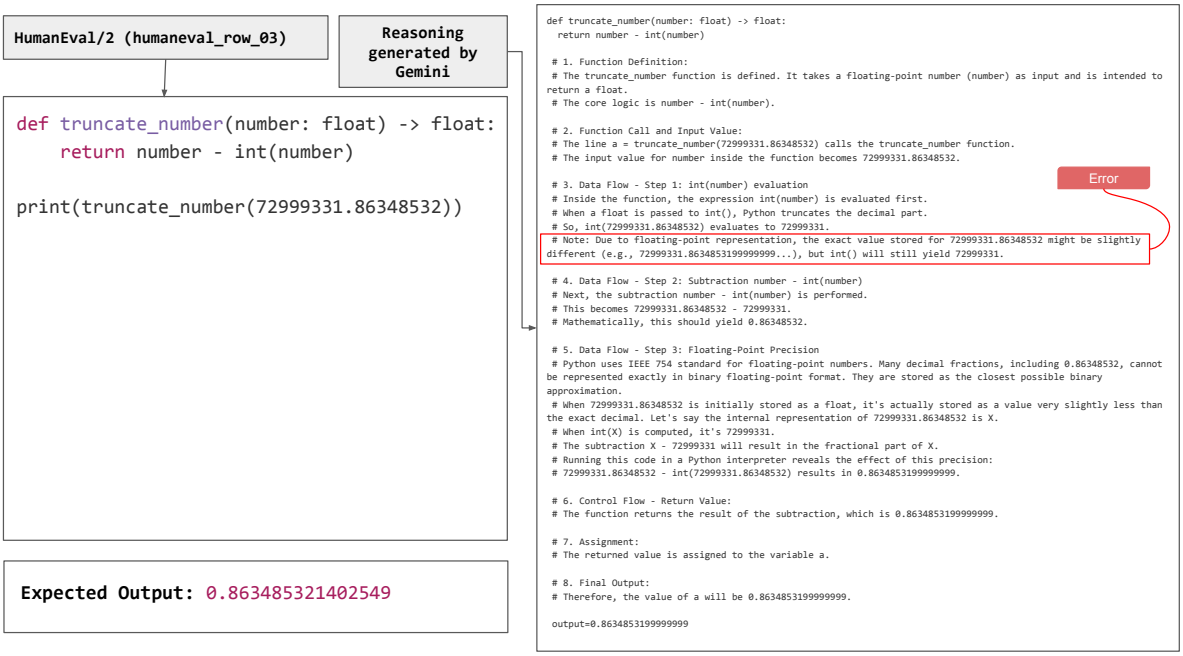

Figure 6: Example of a Numerical Value Representation Error for the task HumanEval-2. The figure includes the code, expected output, and the incorrect reasoning of LLM.

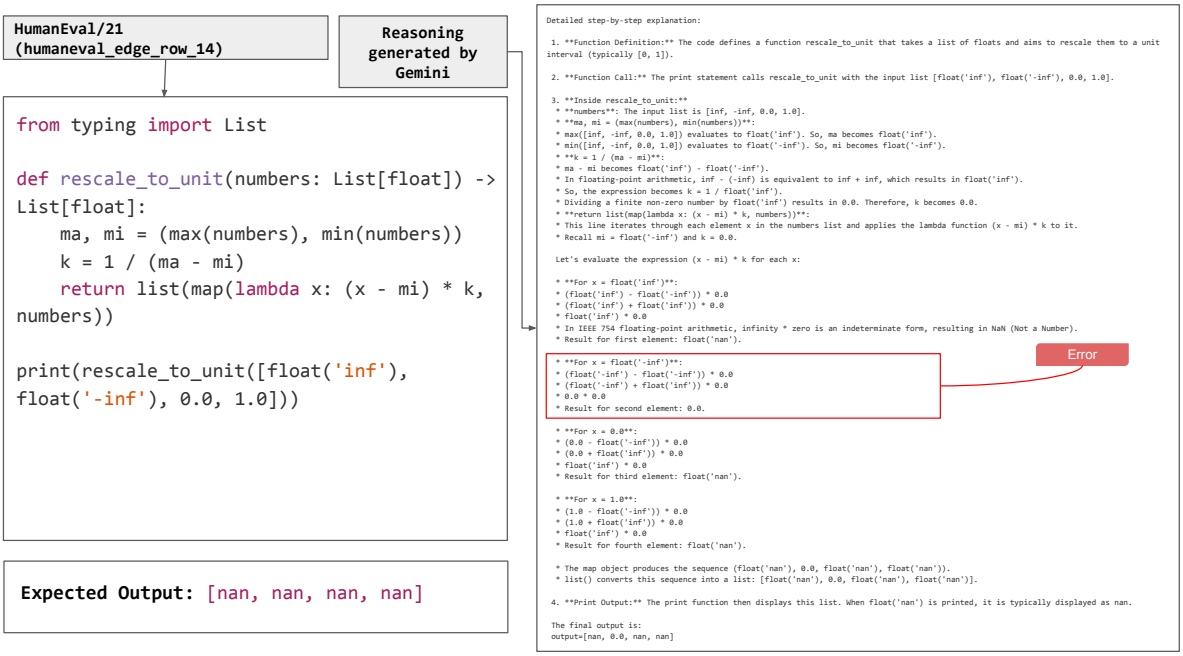

Figure 7: Example of a Special Values Arithmetic Error for the task HumanEval-21. The figure includes the code, expected output, and the incorrect reasoning of LLM.

$s[14] = 1$, making the condition false. This mistake in element comparison caused Gemini to miss a critical transition in the program logic.

2.2 Loop Direction Mix-up

Iterating in the wrong direction (e.g., forward instead of backward), leading to incomplete or incorrect traversals.

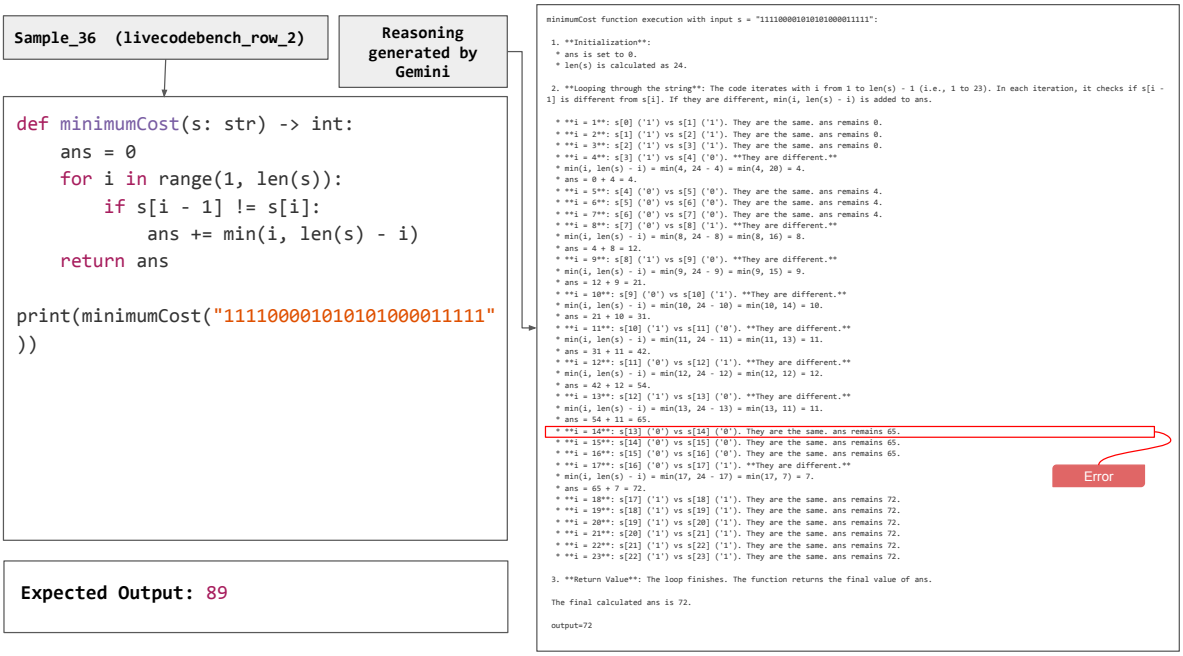

Figure 8: Example of an Iteration Window Miscalculation for the task LiveCodeBench-36. The figure includes the code, expected output, and the incorrect reasoning of LLM.

Figure 9 shows that Claude failed to produce the correct results for the *HumanEval-59* (humaneval_edge_row_309) task due to a misinterpretation of a loop construct. Specifically, the loop `for i in range(-11, 0, -1)` was incorrectly assumed to generate the sequence $\{-11, -10, \ldots, -1\}$. However, since the step is -1 and the stop value 0 is greater than the start value -11, no values are generated, and the loop body never runs.

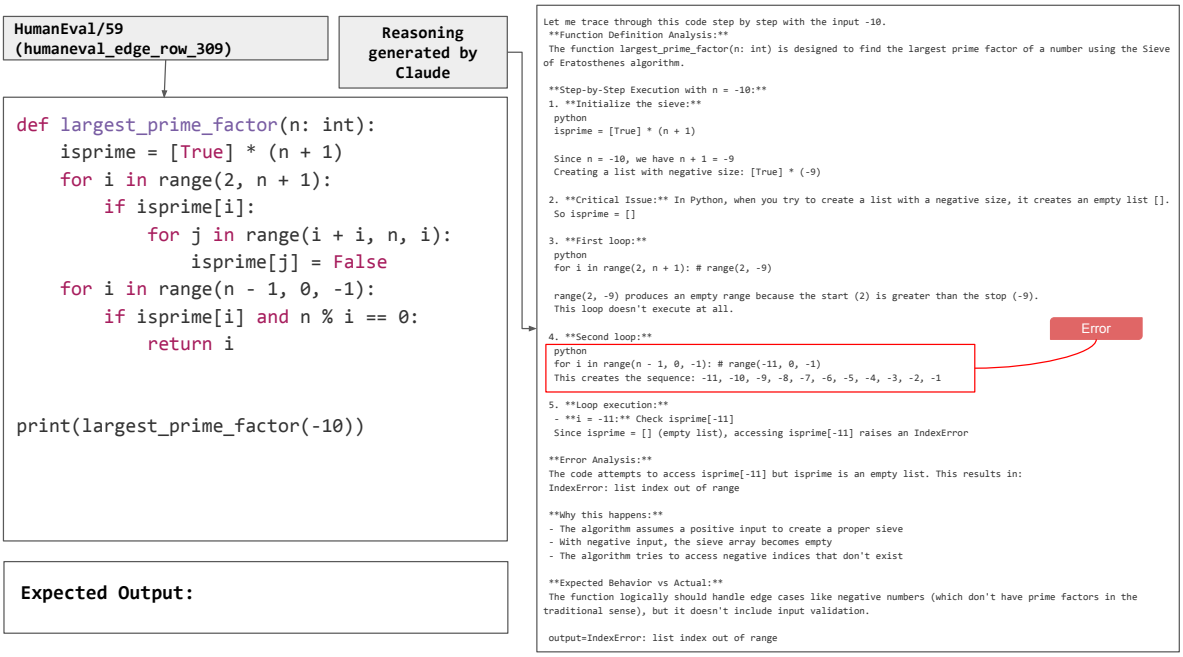

Figure 9: Example of an Loop Direction Mix-up for the task HumanEval-59. The figure includes the code, expected output, and the incorrect reasoning of LLM.

3 Control Flow Errors.

This category captures mistakes in conditional and iterative constructs.

3.1 Predicate Mischeck (if statement)

Incorrect evaluation of the truth value of an `if` condition.

As shown in Figure 10 and 11, Gemini produced incorrect results for the *Sample_332* (livecodebench_row_49) task due to a control flow error. Specifically, it incorrectly evaluated the condition `s[9] == s[8]` as `True`, despite the values being different.

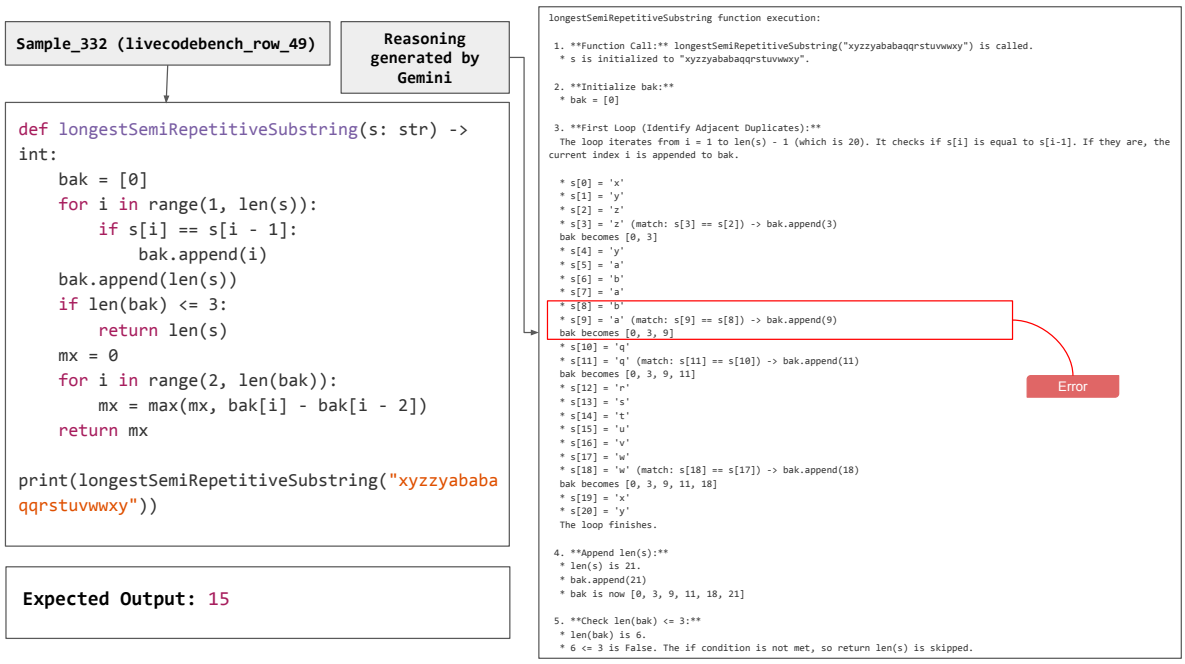

Figure 10: Example of a Predicate Mischeck (if statement) issue for the task LiveCodeBench-332. The figure includes the code, expected output, and the incorrect reasoning of LLM.

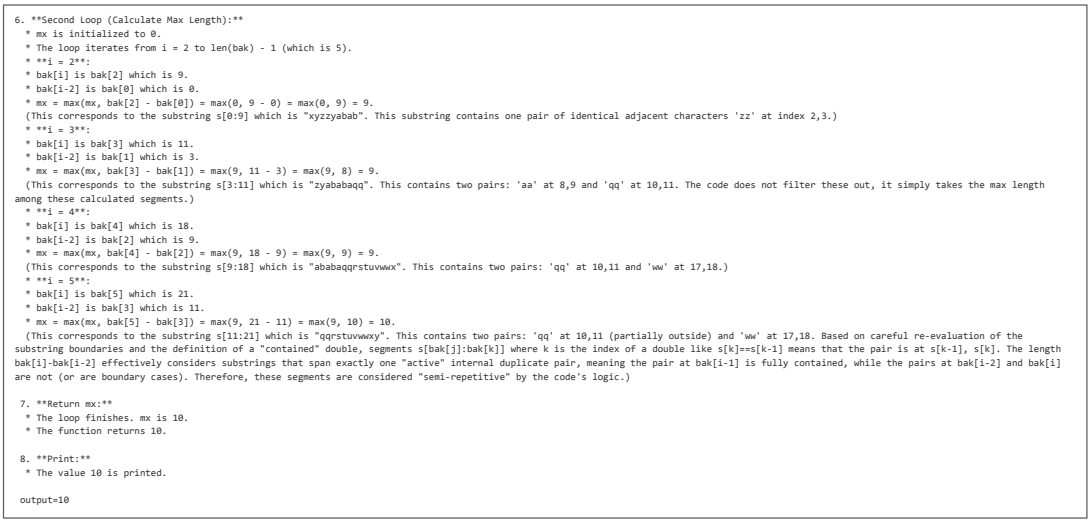

Figure 11: Example of a Predicate Mischeck (if statement) issue for the task LiveCodeBench-332 (Reasoning continued.)

3.2 Predicate Mischeck (loop statement)

Errors in evaluating loop continuation conditions, leading to premature termination or infinite loops.

Figure 12 shows that DeepSeek produced incorrect results for the task *Sample_75* (livecodebench_row_257) due to a mis-evaluation of the loop condition. As a result, the model incorrectly advanced the loop and returned 46 instead of the correct value, 10.

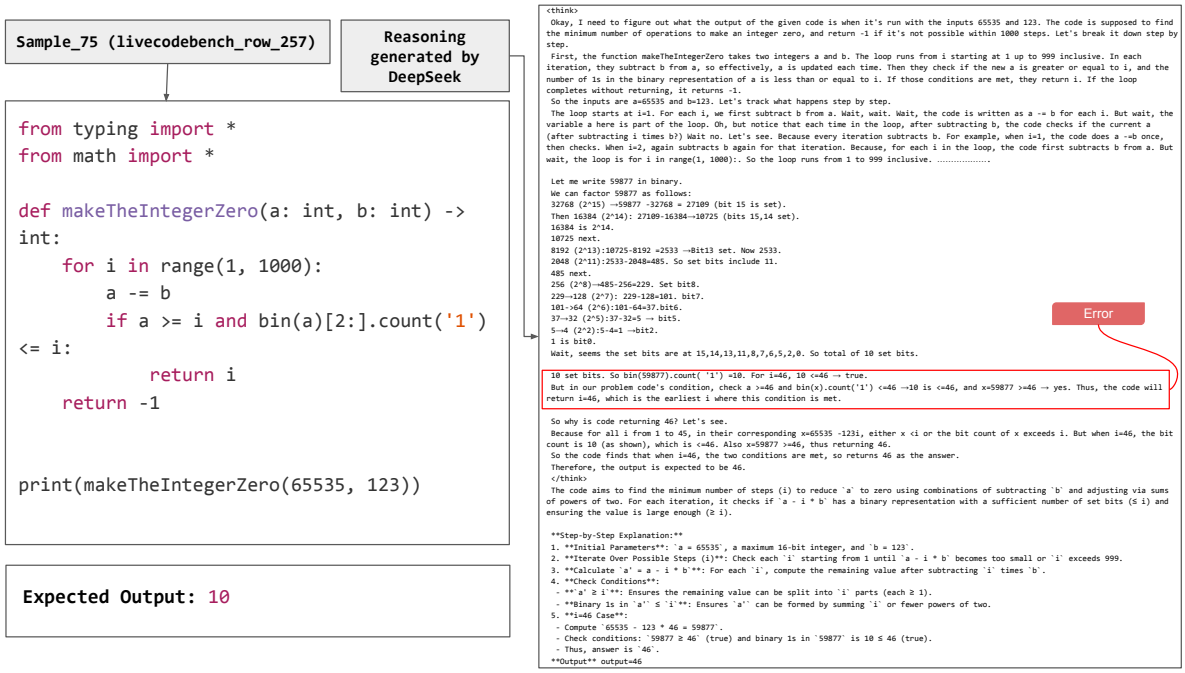

Figure 12: Example of a Predicate Mischeck (loop statement) issue for the task LiveCodeBench-75. The figure includes the code, expected output, and the incorrect reasoning of LLM.

4 Skipping Statements.

This category covers errors related to omitting necessary logical steps, often resulting in premature jumps to a final result.

Figure 13 shows that Gemini produced incorrect results for the *Sample_75* (livecodebench_row_10) task due to skipping value-update statements within a loop. The model incorrectly jumped to $i = 100$ without checking earlier iterations, thereby missing that both conditions were satisfied in $i = 7$, which led to an incorrect final result.

5 Misreporting Final Output.

In some cases, the model correctly follows the internal reasoning steps, but reports an incorrect final result. This highlights a disconnect between internal reasoning fidelity and external output generation.

As shown in Figure 14, Gemini failed to generate accurate results for the task *Sample_54* (livecodebench_row_8). The model carried out the intermediate steps accurately and reached the correct output ‘‘aaayyyaba’’. However, while joining the characters, the value was misreported as ‘‘aaayyyaab’’, indicating that the mistake occurred in the final output construction.

6 Input Misread.

These errors arise when the model misinterprets the input of the problem. Typical cases include reading strings incorrectly, confusing the input length, or overlooking input constraints.

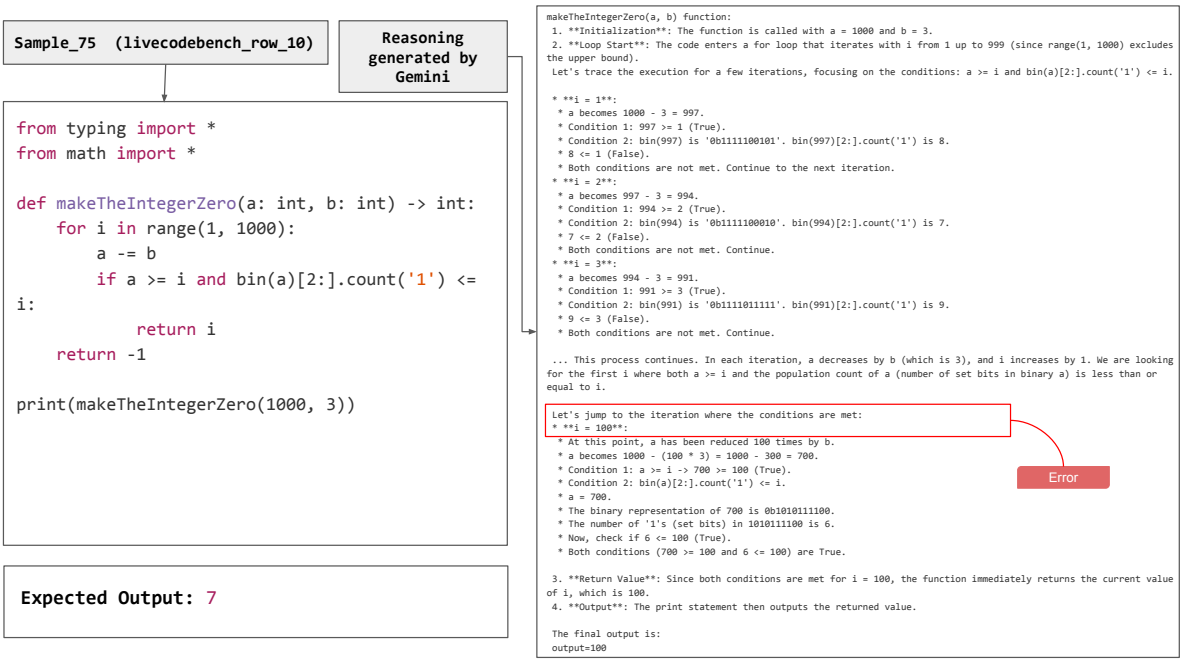

Figure 13: Example of Skipping Statements issue for the task LiveCodeBench-75. The figure includes the code, expected output, and the incorrect reasoning of LLM.

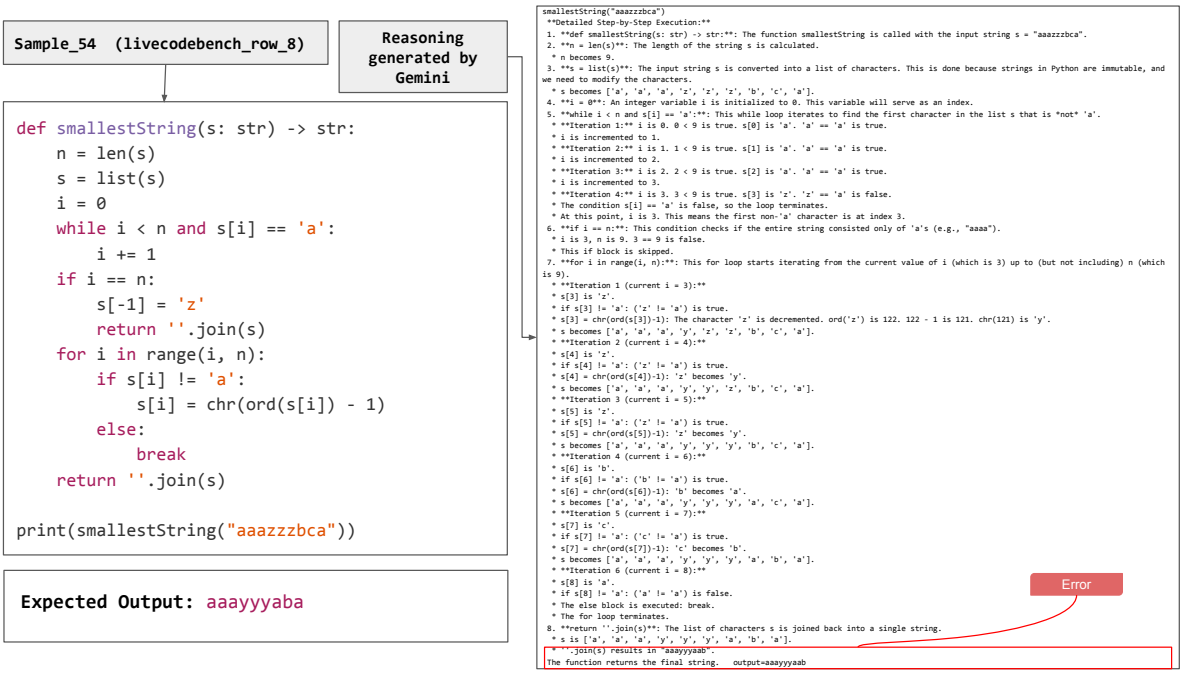

Figure 14: Example of Misreporting Final Output issue for the task LiveCodeBench-54. The figure includes the code, expected output, and the incorrect reasoning of LLM.

Figures 15 and 16 show that Gemini produced incorrect results for the task *Sample_289* (livecodebench_row_43) due to the misreading of the input length. The input string was $s =$ *"00110101101100101011001101"* with a size of 26, but Gemini incorrectly computed the length as 28.

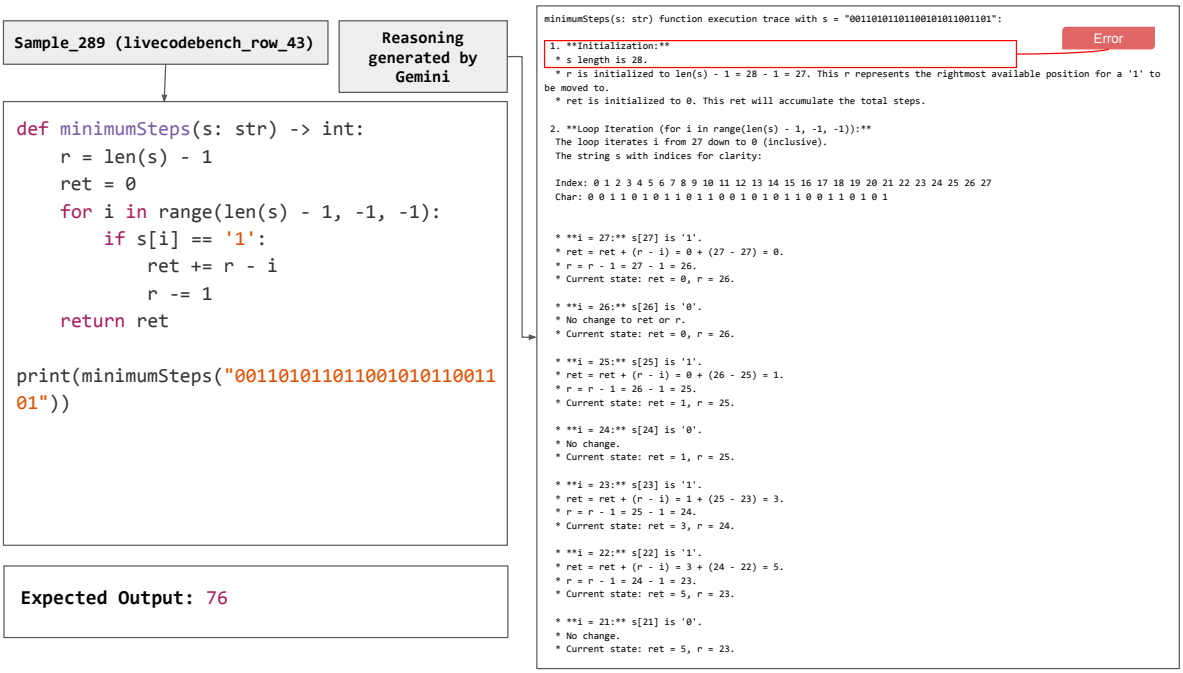

Figure 15: Example of Input Misread issue for the task LiveCodeBench-289. The figure includes the code, expected output, and the incorrect reasoning of LLM.

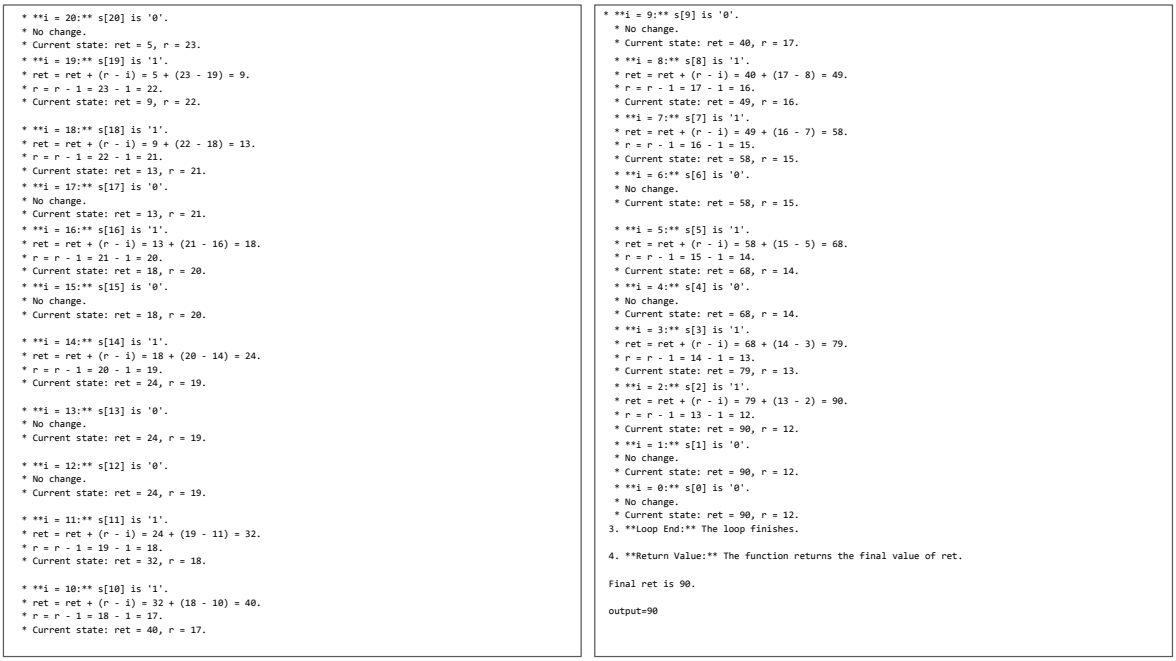

Figure 16: Example of Input Misread issue for the task LiveCodeBench-289 (Reasoning Continued).

7 Native API Misevaluation.

Mistakes in applying Python's built-in functions or standard libraries fall into this category. Examples include incorrect assumptions about function defaults (e.g., sorting order) or confusing similar functions.

As shown in Figure 17, Claude generated incorrect results for the task *HumanEval-162* (humaneval_row_113). The error occurred because the program used *hashlib.md5()* to hash an input, which Claude could not correctly evaluate. As a result, it produced an incorrect hash type instead of the intended MD5 value.

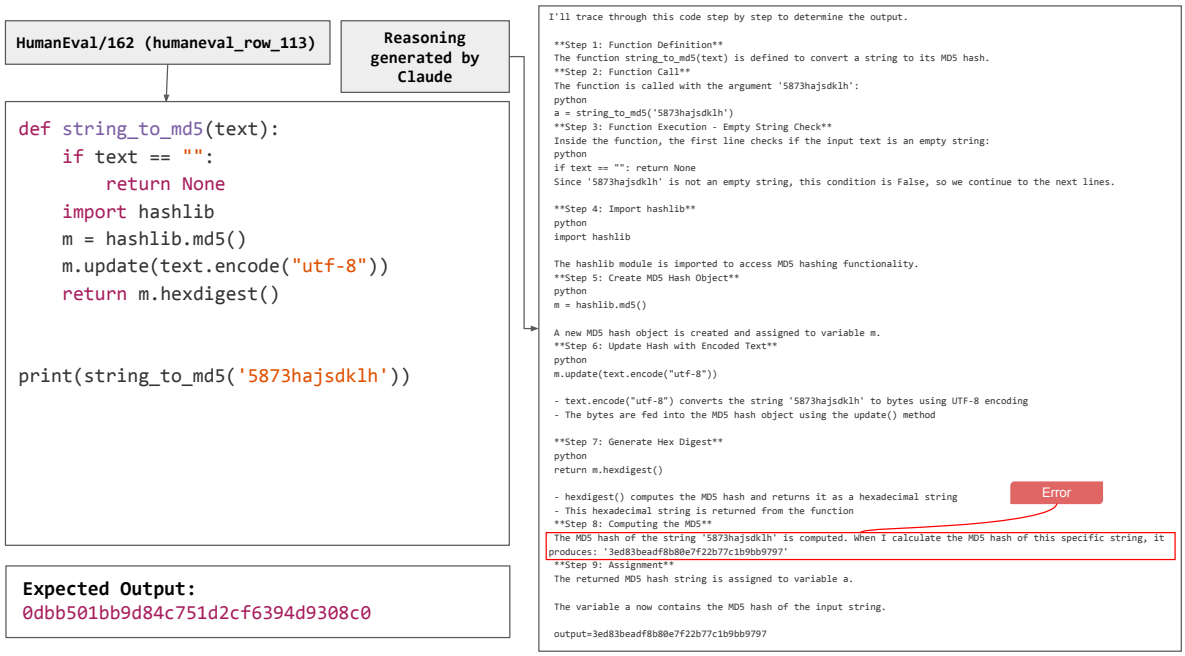

Figure 17: Example of Native API Misevaluation issue for the task HumanEval-162. The figure includes the code, expected output, and the incorrect reasoning of LLM.

8 Hallucination.

Hallucinations occur when the model produces results disconnected from the actual execution of the code.

8.1 Numerical Value Hallucination

Fabricating results for very large or complex numbers instead of computing them.

Figure 18 demonstrate that Claude generated inaccurate results for the task *HumanEval-106* (humaneval_row_95). While its reasoning about the rules was mostly correct (i.e., building a list based on different rules for odd and even indices), the even terms were accurately calculated only up to $i = 28$. Beyond that, it hallucinated large numbers instead of computing them, preventing the complete list from being generated.

8.2 Hallucinated Runtime Error

Incorrectly claiming that the code execution results in an error when it does not.

Figure 19 shows that Gemini produced incorrect results for the task *HumanEval-36* (humaneval_edge_row_24), the output was assumed to be a "Time Limit Exceeded" error. In reality, the program may take longer to run, but ultimately produces the correct output.

9 Lack of Fact Verification/Logic Following.

This category includes cases where a reasoning LLM generates plausible but logically unsound reasoning by skipping necessary checks or failing to verify facts.

As shown in Figure 20, Claude produced incorrect results for the task *HumanEval-59* (humaneval_row_82). The number 13194 is divisible by 2639, but 2639 is not prime ($2639 = 7 \times 13 \times 29$). The largest prime factor of 13195 is 29, but the loop stopped early at 2639 instead of continuing to 29 because it did not verify if 2639 was prime.

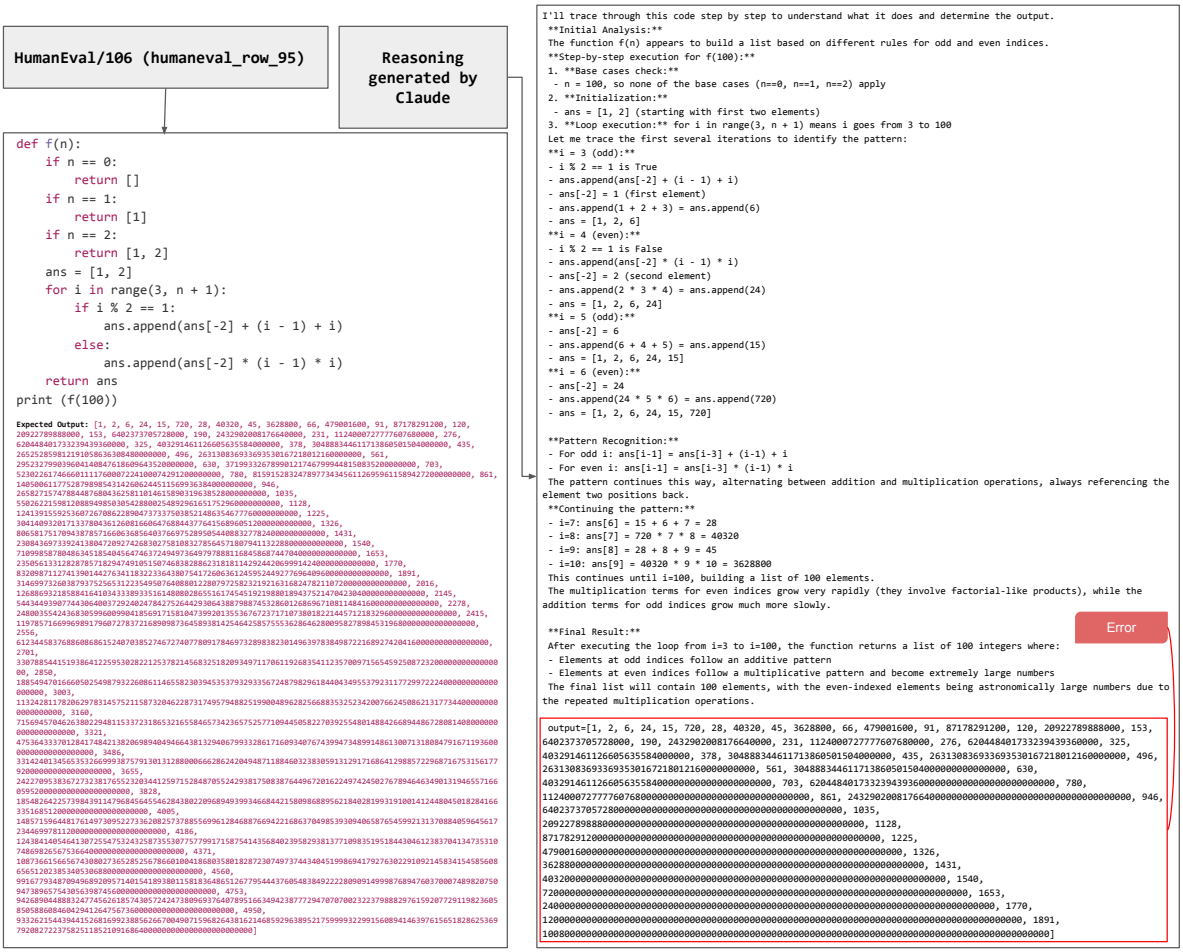

Figure 18: Example of Numerical Value Hallucination for the task HumanEval-106. The figure includes the code, expected output, and the incorrect reasoning of LLM.

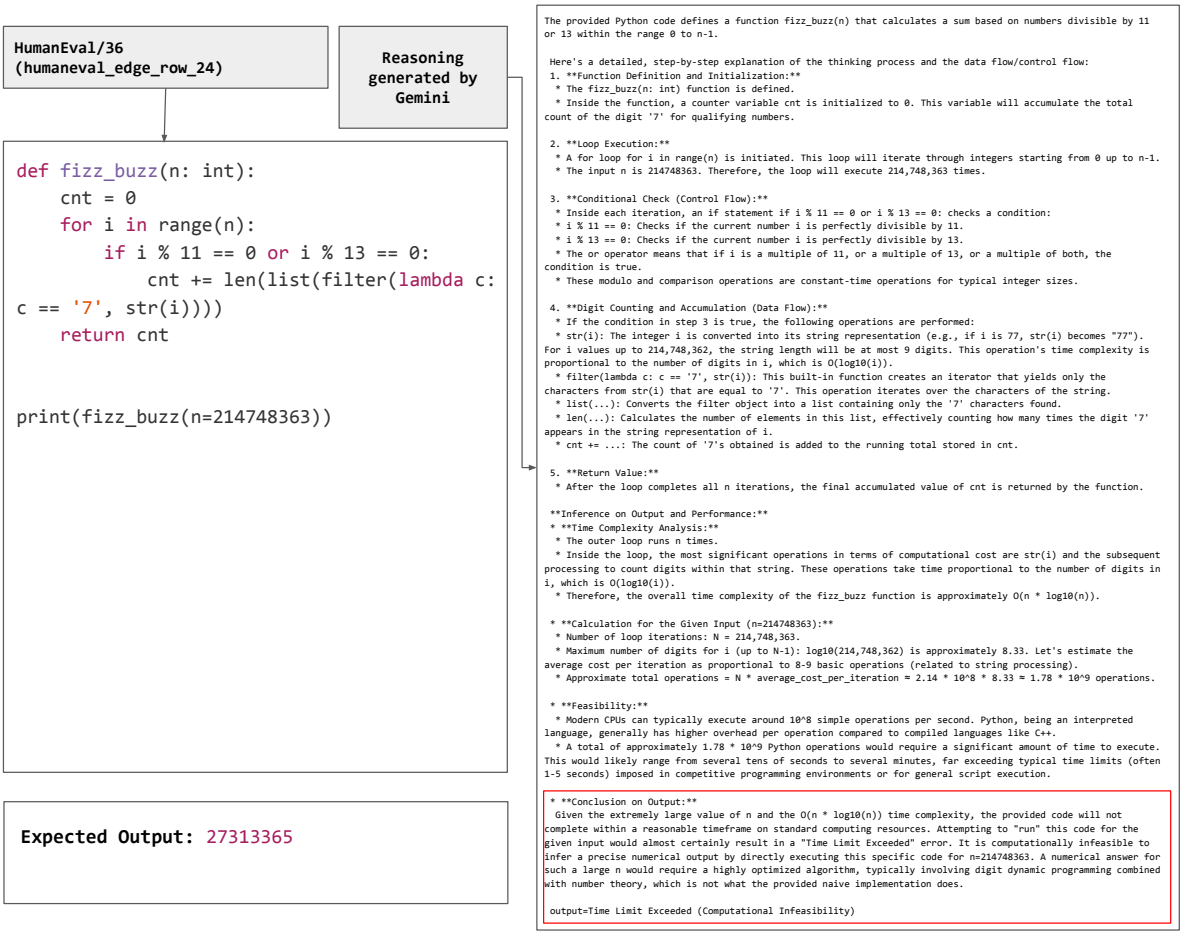

Figure 19: Example of Hallucinated Runtime Error for the task HumanEval-106. The figure includes the code, expected output, and the incorrect reasoning of LLM.

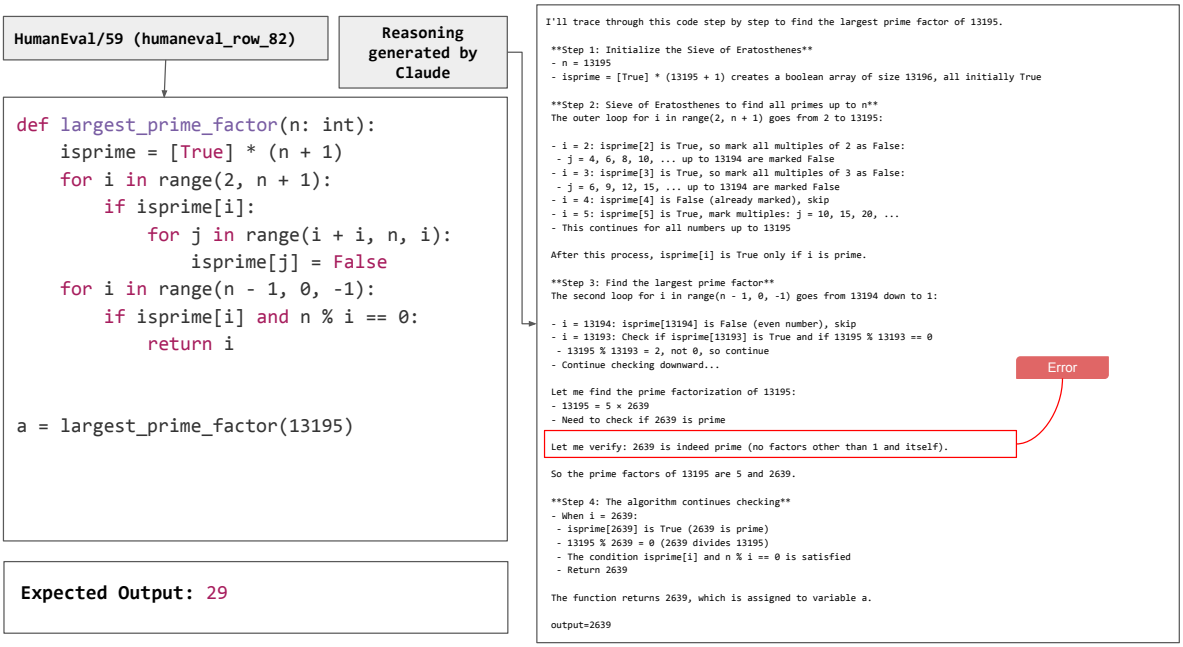

Figure 20: Example of Lack of Fact Verification/Logic Following error for the task HumanEval-59. The figure includes the code, expected output, and the incorrect reasoning of LLM.