# CS598JBR MP Progress Report: Team-1

Divij Gupta, Lukas Getter, Rafay Junaid, Sean Kraemer
University of Illinois Urbana-Champaign
USA
[divij3,lgetter2,rjunaid2,seanmk2]@illinois.edu

## 1 Team Information

Link to the GitHub Repository: CS598JBR-Team-1 Github
Link to the Google Colab Workspace: CS598JBR-Team-1 Google Colab

## 2 MP1

- What is the pass@k metric? Include the formula and explain it [5 points]
  **Answer:**

$$\text{pass@}k = 1 - \frac{C\binom{n-c}{k}}{C\binom{n}{k}}$$

  where:
  - $n$: Total number of generated samples.
  - $c$: Number of correct samples among the $n$ samples.
  - $k$: Number of samples to be considered (top $k$).

  The pass@k metric measures the probability that at least one correct solution appears among the top $k$ generated samples. It is widely used to evaluate the effectiveness of code generation models.

- pass@1 values (after post-processing) under the two different settings [5 points each, total 10 points]
  **Answer:**
  - 0.25: pass@1 under Base model.
  - 0.45: pass@1 under Instruct model.

- A table showing the pairwise comparison of the fail result for each of the 20 programs in your dataset under two different settings, before and after post-processing (problem_ID, base_results, base_results_processed, instruct_results, instruct_results_processed) [40 points]
  **Answer:**

| ID | base | base_processed | instruct | instruct_processed |
|---|---|---|---|---|
| 11 | fail | **pass** | fail | **pass** |
| 147 | fail | fail | fail | fail |
| 107 | fail | fail | fail | fail |
| 70 | fail | fail | fail | fail |
| 152 | **pass** | **pass** | fail | **pass** |
| 125 | fail | fail | fail | fail |
| 160 | fail | fail | fail | fail |
| 90 | fail | fail | fail | fail |
| 142 | fail | fail | fail | **pass** |
| 81 | fail | fail | fail | fail |
| 132 | fail | fail | fail | fail |
| 136 | fail | **pass** | **pass** | **pass** |
| 89 | fail | fail | fail | fail |
| 30 | fail | **pass** | fail | **pass** |
| 122 | fail | fail | fail | **pass** |
| 62 | fail | fail | fail | **pass** |
| 37 | fail | fail | fail | **pass** |
| 93 | fail | fail | fail | fail |
| 56 | fail | **pass** | fail | **pass** |
| 163 | fail | fail | fail | fail |

- Analyze the results in the above table. The analysis should categorize the issues in the initial response and assess whether/how the post-processing addressed them. Do you observe a prevalent pattern of issues? Discuss! [20 points]
  **Answer:**
  The failures in the initial responses from both the base and instruct models can be categorized into three primary types: Syntax and Formatting Errors, Naming Errors, and Semantic/Logic Errors. Our post-processing script proved to be highly effective in mitigating the most prevalent type of issue.
- **Syntax and Formatting Errors** These were the most common issues observed across both models, manifesting as malformed syntax or extraneous code. Common examples include unterminated string literals, missing parentheses, incorrect indentation, and the generation of multiple functions or test cases within a single response. Our robust post-processing step was designed to address these issues. A classic example of this is the response for HumanEval/11. The raw output provides multiple redundant functions, which causes the test to fail.

```python
from typing import List

def string_xor(a: str, b: str) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor('010', '110')
    '100'
    """
    return ''.join([str(int(a[i]) ^ int(b[i])) for i in range(len(a))])

def string_xor_list(a: str, b: List[str]) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor_list('010', ['110', '101', '011'])
    '100'
    """
    return ''.join([str(int(a[i]) ^ int(b[i])) for i in range(len(a))])

def string_xor_list_list(a: List[str], b: List[str]) -> str:
    """ Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor_list_list(['010', '110'], ['110', '101', '011'])
    '100'
```

```
    """
    return ''.join([str(int(a[i]) ^ int(b[i])) for
      i in range(len(a))])

def string_xor_list_list_list(a: List[str], b:
     List[str], c: List[str]) -> str:
    """ Input are two strings a and b consisting
     only of 1s and 0s.
    Perform binary XOR on these inputs and return
     result also as a string.
    >>> string_xor_list_list_list(['010',
```

1: Raw Response (Base) - HumanEval/11

By focusing the response down to the first and only function, which the prompt was expecting, the post-processing script correctly resolves the issue. This causes the same prompt to go from failed to pass.

```
def string_xor(a: str, b: str) -> str:
    """ Input are two strings a and b consisting
     only of 1s and 0s.
    Perform binary XOR on these inputs and return
     result also as a string.
    >>> string_xor('010', '110')
    '100'
    """
    return ''.join([str(int(a[i]) ^ int(b[i])) for
      i in range(len(a))])
```

2: Processed Response (Base) - HumanEval/11

This pattern of post-processing fixing the initial LLM response is a prevalent trend across the dataset. Other problems that exhibit this pattern include 152, 142, 136, 30, 122, 62, 37, and 56.

- **Naming and Semantic Errors** The model responses for these problems were largely on the right track conceptually, but they would make references to undeclared or incorrectly named variables. Unlike syntactic issues, post-processing does not correct these errors, as they require a deeper understanding of the code's logic.
  Here is an example from HumanEval/147:

```
def get_max_triples(n):
    """
    You are given a positive integer n. You have
     to create an integer array a of length n.
        For each i (1 <= i <= n), the value of a[i
    ] = i * i - i + 1.
        Return the number of triples (a[i], a[j],
     a[k]) of a where i < j < k,
    and a[i] + a[j] + a[k] is a multiple of 3.

    Example :
        Input: n = 5
        Output: 1
        Explanation:
        a = [1, 3, 7, 13, 21]
        The only valid triple is (1, 7, 13).
    """
    a = [i * i - i + 1 for i in range(1, n + 1)]
    count = 0
    mod_count = [0] * 3
    for num in a:
        mod_count[num % 3] += 1
    count += mod_count[0] * (mod_count[0] - 1) * (
     mod_count[0] - 2) // 6
    count += mod_count[1] * (mod_count[1] - 1) *
     mod_count[2] // 2
    count += mod_count[2] * (mod[2] - 1) * (
     mod_count[2] - 2) // 6
    return count

```

3: Raw Response (Instruct) - HumanEval/147

The model makes a reference to the variable 'mod' on line 22 when it should have used 'mod_count'. This type of error is not fixable by simple string manipulation, and as a result, the processed response also fails. This problem is also prevalent across several other problems/model variants such as:

  - HumanEval/107: instruct (unprocessed) fails with message: "name 'even' is not defined."
  - HumanEval/70: instruct models fail with message "name 'strangeort_list' is not defined."
  - HumanEval/132: instruct models fail with message "name 'isis_nested' is not defined."

- **Prevalent Pattern** The most significant pattern observed is the effectiveness of post-processing in addressing a wide range of common syntactic flaws. While the raw outputs frequently suffer from extraneous functions and syntax errors, our post-processing script successfully cleans these responses, leading to a substantial increase in passing test cases. However, this method is limited to structural issues and does not address fundamental semantic misunderstandings, such as incorrect variable names or logical errors, which remain a key challenge in LLM-based code generation. The Instruct model demonstrates a higher overall pass rate, but both models exhibit a similar distribution of failure types, suggesting that the underlying challenges in generating valid and correct code are consistent.

## 3 MP2

TBA

## 4 MP3

TBA