CS598JBR MP Progress Report: Team-16

Jaakko Hyöty, Qiantong Zhong, Yun-Yang Huang, Vincent Ma
University of Illinois Urbana-Champaign
[jhyty2,qz23,yh76,yuxuanm4]@illinois.edu

1 TEAM INFORMATION

Link to the GitHub Repository: https://github.com/qz0610/CS598JBR-Team-16

Link to the Google Colab Workspace for MP1: https://colab.research.google.com/drive/1VFAFFNjRoS0ikanAhHt1v2aU1yULXllZ
Link to the Google Colab Workspace for MP2: https://colab.research.google.com/drive/1dDAZ59VKFHKehnX_Xv11uHTIgGqaquSi Link to the Google Colab Workspace for MP3: https://colab.research.google.com/drive/1prQGtl1piOl9H01PEwTTwA5qxlFRM1S-

2 MP1

- **What is Pass@k Metric?**
  Pass@k measures the proportion of tasks for which at least one correct answer is found in the top k results returned by a model.
- **pass@1 values under the two different settings**
  - DeepSeekCoder-6.7b-base: 0.15, which pass 3 out of 20 tasks
  - DeepSeekCoder-6.7b-instruct: 0.20, which pass 4 out of 20 tasks
- **A table showing the pairwise comparison of the pass/-fail result for each of the 20 programs in your dataset under two different settings**

Table 1: Pairwise Comparison of Pass/Fail Results for Each HumanEval Task

HumanEval Task	Base Setting	Instruct Setting
HumanEval/24	Fail	Pass
HumanEval/87	Fail	Pass
HumanEval/113	Fail	Fail
HumanEval/108	Fail	Fail
HumanEval/65	Fail	Fail
HumanEval/136	Fail	Fail
HumanEval/112	Fail	Fail
HumanEval/8	Pass	Pass
HumanEval/38	Fail	Fail
HumanEval/153	Fail	Fail
HumanEval/154	Fail	Fail
HumanEval/36	Fail	Pass
HumanEval/93	Fail	Fail
HumanEval/103	Fail	Fail
HumanEval/18	Pass	Fail
HumanEval/62	Fail	Fail
HumanEval/137	Pass	Fail
HumanEval/92	Fail	Fail
HumanEval/45	Fail	Fail
HumanEval/56	Fail	Fail

3 MP2

MP2-task1

- **To what extent can DeepSeekCoder reason about code execution? Report the success rate (calculate the success rate as the number of programs that LLM correctly predicts their output divided by 20 (the total number of programs in your experiments)) of the prompt crafting here.**
  - Vanilla: 6/20 = 0.3
  - Crafted: 13/20 = 0.65
- **What factor impacts the model's performance, i.e., when it can and cannot (you can explore the presence of specific programming constructs in the code, complex logic operations, nested constructs, etc.)?**
  - The model struggles sometimes to place the prediction between the [Output] and [/Output] tags. This actually leads to a lot of incorrectly predicted results, although the reasoning might be correct. We tried multiple different styles to emphasize the importance of the tags, but it seemed to confuse the model more.
  - The model also struggled with following manipulation of lists, and had difficulties predicting outputs when input consisted of a list.

  - Sometimes the model missed crucial information, such as .swapcase() and returned a prediction in uppercase when it would have been correct in lowercase.
  - Surprisingly, the models performed relatively well even in complex loops with strings as input. Our guess is that loops are well present in training datasets.

- **What changes to the vanilla prompt help the model to perform better on the task?**
  - The model benefited from a few changes, but often when trying to provide more information, the model got more confused. We found that a simple prompt structure helped avoid confusion and in our prompt, we always split our sub-sections with '###'. E.g. "### Function comments:", "### Function definition:", and "### Instructions:".
  - In "### Function comments:" we provided the helpful comments that were present in the HumanEval prompts, but we replaced keywords such as "You " and "Create " with "The function" and "" respectively, in order to not confuse the model with the task.
  - The comments helped a lot. However, some of the comments included examples that were part of the tests, and they possibly allowed the model to more easily predict the return value without logically thinking through the function.

- **A table showing the values for is_correct from vanilla prompting and prompt crafting**

MP2-task2

- **To what extent can DeepSeekCoder produce reasonable unit tests for a given code snippet? Report the percentage of generated unit tests that pass or fail (have a table where each row shows (1) the total number of generated tests, (2) the number of tests that pass, and (3) the number of tests that fail per program)**
  Please see Table 3 and Table 4 for results.

Table 2: Each row shows the values for is_correct from vanilla prompting and prompt crafting

HumanEval Task	vanilla	crafting
HumanEval/24	true	false
HumanEval/87	true	true
HumanEval/113	false	true
HumanEval/108	false	true
HumanEval/65	false	false
HumanEval/136	false	false
HumanEval/112	true	true
HumanEval/8	false	true
HumanEval/38	false	false
HumanEval/153	false	true
HumanEval/154	false	true
HumanEval/36	false	true
HumanEval/93	false	false
HumanEval/103	true	true
HumanEval/18	true	true
HumanEval/62	false	true
HumanEval/137	false	true
HumanEval/92	true	false
HumanEval/45	false	false
HumanEval/56	false	true

Table 3: Vanilla Prompting, each row shows the total number of generated tests, the number of tests that pass/fail per program, and the percentages of passed tests

HumanEval Task	number of tests	#pass	#fail	percentage(%)
HumanEval/24	10	3	7	30
HumanEval/87	5	3	2	60
HumanEval/113	4	1	3	25
HumanEval/108	4	2	2	50
HumanEval/65	4	3	1	75
HumanEval/136	4	2	2	50
HumanEval/112	4	2	2	50
HumanEval/8	3	2	1	67
HumanEval/38	5	3	2	60
HumanEval/153	3	0	3	00
HumanEval/154	5	3	2	60
HumanEval/36	10	4	6	40
HumanEval/93	6	1	5	17
HumanEval/103	4	2	2	50
HumanEval/18	6	4	2	67
HumanEval/62	5	4	1	80
HumanEval/137	6	6	0	100
HumanEval/92	3	3	0	100
HumanEval/45	4	4	0	100
HumanEval/56	1	0	1	0

Table 4: Crafted Prompting, each row shows the total number of generated tests, the number of tests that pass/fail per program, and the percentages of passed tests

HumanEval Task	number of tests	#pass	#fail	percentage(%)
HumanEval/24	5	3	2	60
HumanEval/87	5	4	1	80
HumanEval/113	5	1	4	20
HumanEval/108	5	2	3	40
HumanEval/65	5	1	4	20
HumanEval/136	5	1	4	20
HumanEval/112	5	4	1	80
HumanEval/8	5	3	2	60
HumanEval/38	5	2	3	40
HumanEval/153	5	2	3	40
HumanEval/154	5	3	2	60
HumanEval/36	5	2	3	40
HumanEval/93	5	1	4	20
HumanEval/103	5	1	4	20
HumanEval/18	5	4	1	80
HumanEval/62	5	2	3	40
HumanEval/137	5	3	2	60
HumanEval/92	5	4	1	80
HumanEval/45	3	3	0	100
HumanEval/56	5	4	1	80

- **What are the reasons for test failures, if you observe any? For the cases in which there are no test failures, can you confidently say the program is bug-free?**
  The failures often result from five reasons:
  - The response incorrectly includes the implementation of the function to be tested causing the testing to not run as intended.
  - The response includes the function implementation and the automated coverage results provide a really low number as the test file is not supposed to include the actual function implementation.
  - The response includes calls to the function to be tested, but incorrectly spells the function name.
  - The assertions do not take into account mathematical issues, such as division by zero errors.
  - The input-output pairs are not logically correct.

  The coverage for many is 100% since the functions are often just a few lines without multiple branches. However, 100% coverage does not mean bug-free. Often the tests are not even logically correct and having 100% passing tests seems very difficult with the model. This is a challenging task for the LLM as it needs to predict the output as in task 1, but now with an input value of its own.
- **What changes to the vanilla prompt help the model to perform better on the task? Please include a table showing the pairwise comparison of test suite coverage values per each program for vanilla prompting and prompt crafting (have a table where each row shows the coverage results from vanilla prompting and prompt crafting)**
  Couple of changes help the model:
  - Simplifying the prompt. We noticed that by removing the "You are an AI assistant…" text from the beginning of the prompt, the model seems less confused. Due to the smaller model size, the model is probably able to produce more syntactically correct code.
  - We provided the HumanEval prompt and the canonical solution. In some cases, this included examples for tests that appeared to help the model to reason better on the input-output pairs.
  - We provide the model the beginning of the response with the correct from "module" import "function" statement to promote the correct import of the function to be tested.
  - We separately state the name of the function to promote the function being called with the correct name.
  - Finally, we limited the number of tests to 5. We noticed that if the model was given a task to generate a large number of tests, the model might lose track and quit abruptly or continue to repeat a certain character until it hit the max_length of the response.

  Please see Table 5 for the comparisons between the coverage results from vanilla prompting and prompt crafting.

Table 5: Coverages for Vanilla Prompting and Crafted Prompting

HumanEval Task	Coverage (Vanilla)	Coverage (Crafted)
HumanEval/24	100.0	100.0
HumanEval/87	100.0	100.0
HumanEval/113	100.0	100.0
HumanEval/108	100.0	100.0
HumanEval/65	100.0	100.0
HumanEval/136	100.0	100.0
HumanEval/112	100.0	100.0
HumanEval/8	25.0	100.0
HumanEval/38	83.33	100.0
HumanEval/153	100.0	80.0
HumanEval/154	100.0	100.0
HumanEval/36	100.0	100.0
HumanEval/93	100.0	100.0
HumanEval/103	100.0	100.0
HumanEval/18	100.0	100.0
HumanEval/62	100.0	100.0
HumanEval/137	100.0	100.0
HumanEval/92	100.0	83.33
HumanEval/45	100.0	100.0
HumanEval/56	11.1	100.0

- **Do you observe any relationship between code reasoning and test generation, or specifically, generating tests that pass? Please elaborate and, ideally, perform a correlation analysis.**
  Code reasoning and test generation seem to be positively

correlated, i.e. improved code reasoning (achieved by more carefully crafted prompting) increases the chance of generating tests that pass.
Furthermore, by comparing values from table 2 and table 4, we see that for example with HumanEval/65, /93, and /136 the models were unable to predict outputs and also had poor test passing rates. However, this is not always the case as can be seen with HumanEval/45, for which models could not predict output correctly, but in task 2 models achieved 100% passing rate. Since the prompts were quite similar, this suggests that small differences in prompting can make a big difference in inference results.

4 MP3

MP3-task1

- **What changes to the vanilla prompt help the model to perform better on this task? Please include a table showing the pairwise comparison of the is_correct value per each program for vanilla prompting and prompt crafting (have a table where each row shows vanilla prompting and prompt crafting results per each program). If a translation results in a compilation error and does not reach the test execution phase, please mark its outcome as "False (compilation issue)" in this table.**
  As can be seen from Table 6, the main point for the vanilla prompt was to translate code that would compile. Often the generated translation had syntactic issues and the model was not able to even keep in focus what the correct method name was. With a higher compilation rate we could achieve more correct translations and, thus, we focused our efforts in providing a prompt that would result in a compilable translation. For this we:
  - Provided emphasis on the correct output format. The model by default wanted to always provide the translated code enclosed in ``` java ... ``` despite being instructed to use tags [Java Start] and [Java End]. To emphasize the correct format, we added an example format to the prompt:

    ```
    Example format:
    [Java Start]
    public class Main {
     public static void main(String[] args) {
        System.out.println("Hello, world!");
        }
    }
    [Java End]
    ```

  - We also used the phrase "code snippet instead of "following code", as this improved compilation rates. It seemed that the LLM was able to understand this phrase better.
  - We avoided any other instructions. We tried providing more context from the original HumanEval prompt, but this seemed to confuse the model more and the translations were again less likely to compile correctly.

Table 6: Pairwise comparison of the is_correct value per each program for vanilla prompting and prompt crafting

HumanEval Task	is_correct (Vanilla)	is_correct (Crafted)
HumanEval/24	True	True
HumanEval/87	False(compilation issue)	False
HumanEval/113	False(compilation issue)	False
HumanEval/108	False(compilation issue)	False
HumanEval/65	False(compilation issue)	True
HumanEval/136	False(compilation issue)	False(compilation issue)
HumanEval/112	False(compilation issue)	False
HumanEval/8	False(compilation issue)	False
HumanEval/38	False(compilation issue)	True
HumanEval/153	False(compilation issue)	False
HumanEval/154	False(compilation issue)	False
HumanEval/36	False(compilation issue)	False
HumanEval/93	False(compilation issue)	False
HumanEval/103	False(compilation issue)	False
HumanEval/18	False(compilation issue)	True
HumanEval/62	False	False
HumanEval/137	False	False
HumanEval/92	False(compilation issue)	False
HumanEval/45	True	True
HumanEval/56	True	True

- **Discuss the circumstances and examples of when the model can or cannot correctly translate the code. What properties in the source code make it more challenging for the LLM to translate it into target code? Do you observe a persistent pattern? What is the most prevalent cause of translation failure in your dataset, test failure, or compilation error?**
  - In some instances complex single line operations seem to be confusing. Could be due to lack of natural language.

    Example: HumanEval/87
    ```
    Python:
    coords = [(i, j) for i in range(len(lst)) \
              for j in range(len(lst[i])) \
              if lst[i][j] == x] ...
    Translation: - function loses track of the
                   correct return type
    ```

  - In some instances the translated function does not include validation of inputs, unlike in the original function, leading to errors when testing with floats.

    Example: HumanEval/92
    ```
    Python:
    if isinstance(x,int) and isinstance(y,int) \
    and isinstance(z,int) ...
    ```

```
Translation:
public static boolean anyInt(int x, int y, int z) ...
```

– In some instances the translations followed the naming convention of the original python code, but the tests expected a different naming convention.

Example: HumanEval/103

```
Python:
def rounded_avg(n, m)
Translation:
public static String rounded_avg(int n, int m)
Test call:
s.roundedAvg(1, 5)
```

– In some instances the translation disregards the original function name.

Example: HumanEval/62

```
Python:
def derivative(xs: list)
Translation:
public static int[] calculateDerivative(int[] xs)
```

Thus, the most prevalent issues are translation errors, but test failures are present as well. Our prompt engineering significantly improved compilation of translated code, and in the end there was only one translation that did not compile, as opposed to 15 non-compilable translations with the vanilla prompt. The testing method is not either perfect, as the translation might be correct, but the naming convention might differ trivially. Overall, we notice that the model might struggle more with complex single line nested computations with limited natural language. Furthermore, the translations might omit crucial input validations. However, the naming of functions is one of the most persistent issues, as not only does the method name need to match the original python function, it needs to be aligned with the naming convention present in test files.

- **Compare the translation success rate in the vanilla prompting and prompt crafting. You can compare the success rate by dividing the number of programs that are successfully translated using each approach by 20, i.e., the total number of programs.**
  – Vanilla success rate: 3/20 = 15%
  – Crafted success rate: 6/20 = 30%

MP3-task2

- **What changes to the vanilla prompt help the model to perform better on the task? Please include a table showing the pairwise comparison of the is_correct value per each program for vanilla prompting and prompt crafting (have a table where each row shows the results from vanilla prompting and prompt crafting)**

(1) The crafted prompt adds the function description and explicitly states the goal of evaluating whether the function implementation follows the given description, whereas the vanilla prompt asks about correctness ambiguously. This precision helps the model base its reasoning to the logic defined in the function description.
(2) The crafted prompt removes some details like the assistant's identity (e.g. "You are an AI programming assistant"), making the prompt more concise.
(3) By isolating the function description and function implementation into labeled sections, the crafted prompt organizes the input more clearly. This structured format ensures the model understands that the function description is the primary reference for assessing the implementation.

Table 7: Correctness of Predictions for Vanilla and Crafted Prompting

HumanEval Task	is_correct (Vanilla)	is_correct (Crafted)
HumanEval/24	True	False
HumanEval/87	False	True
HumanEval/113	False	True
HumanEval/108	False	True
HumanEval/65	False	True
HumanEval/136	False	True
HumanEval/112	False	False
HumanEval/8	False	False
HumanEval/38	False	True
HumanEval/153	False	True
HumanEval/154	False	True
HumanEval/36	False	True
HumanEval/93	False	False
HumanEval/103	False	True
HumanEval/18	False	True
HumanEval/62	False	False
HumanEval/137	False	True
HumanEval/92	False	True
HumanEval/45	False	False
HumanEval/56	False	False

- **Compare the success rate in vanilla prompting and prompt crafting: the number of programs that LLM correctly predicts their output divided by 20 (the total number of programs in your experiments)**
  – Vanilla success rate: 1/20 = 5%
  – Crafted success rate: 13/20 = 65%

- **Discuss the circumstances and examples of when the model can or cannot detect the bugs.**
  Circumstances when the model can detect bugs:
  (1) The function description is simple and clear, and the implementation directly contradicts the description:

  Example: HumanEval/45. The description states:

> Given the length of a side (a) and height (h), return the area of a triangle.

The buggy implementation:

```
        def triangle_area(a, h):
            return a * h / 0.5  # Incorrect formula

```

Listing 1: Buggy Code for HumanEval/45

The formula for the area of a triangle is $0.5 \times a \times h$. This implementation incorrectly divides by 0.5, leading to an obvious contradiction.

(2) The implementation has multiple missing components required by the description.

Example: HumanEval/92. The description specifies:

> Return True if one of the numbers is equal to the sum of the other two, and all numbers are integers.

The buggy implementation:

```
    def any_int(x, y, z):
        if isinstance(x, int) and isinstance(y, int) and isinstance(z, int):
            if (x + y == z) or (y + z == x):
    # Missing case: z + x == y
                return True
        return False

```

Listing 2: Buggy Code for HumanEval/92

The condition fails to account for all permutations of the sum condition (e.g., `z + x == y` is missing) and whether all numbers are integers. The model identifies one of the missing condition and flags the code as buggy.

(3) The implementation has simple logical errors.

Example: HumanEval/36. The description specifies:

> Return the number of times the digit 7 appears in integers less than n that are divisible by 11 or 13.

The buggy implementation:

```
    def fizz_buzz(n: int):
        ns = []
        for i in range(n):
            if i % 11 == 0 and i % 13 == 0:  # Incorrect: "and" instead of "or"
                ns.append(i)
        s = ''.join(list(map(str, ns)))
        ans = 0
        for c in s:
            ans += (c == '7')
        return ans

```

Listing 3: Buggy Code for HumanEval/36

Here, the logical condition uses and, meaning only numbers divisible by both 11 and 13 are included. However, the description specifies "or," so the function has obvious logical error.

Circumstances when the model cannot detect bugs:

(1) The task has subtle or implicit edge cases.

Example: HumanEval/93. The description specifies:

> Encode a message by swapping case and replacing vowels with the letter two positions ahead.

The buggy implementation:

```
        def encode(message):
            vowels = "aeiou"
            vowels_replace = dict([(i, chr(ord(i) + 2)) for i in vowels])
            message = message.swapcase()
            return ''.join([vowels_replace[i] if i in vowels else i for i in message])

```

Listing 4: Buggy Code for HumanEval/93

The function does not account for uppercase vowels but the model fails to identify this bug due to the case not explicitly stated in the function description.

(2) The function description is unclear about the exact requirement for the task.

Example: HumanEval/56. The description specifies: brackets is a string of "<" and ">". return True if every opening bracket has a corresponding closing bracket.

```
    def correct_bracketing(brackets: str):
        depth = 0
        for b in brackets:
            if b == ">":
                depth += 1
            else:
                depth -= 1
            if depth < 0:
                return False
        return depth == 0

```

Listing 5: Buggy Code for HumanEval/56

The buggy code requires > appears before <, which is the opposite of a valid bracket. However, the description does not clarify the requirement for a bracket to be valid, and the model thus fails to catch the bug.

(3) Sometimes the model fails to catch mistakes involving simple math.

Example: HumanEval/8. The description specifies:

> For a given list of integers, return a tuple consisting of the sum and product of all the integers in the list. If the list is empty, the sum should be 0 and the product should be 1.

The buggy implementation:

```
    def sum_product(numbers: list) -> tuple:
        sum_value = 0
        prod_value = 0  # Incorrect initialization

        for n in numbers:
            sum_value += n
            prod_value *= n  # Incorrect logic
        return sum_value, prod_value
```

```
9
```

Listing 6: Buggy Code for HumanEval/8

The buggy code initializes product value to 0, which makes the product value always stays 0. Sometimes LLMs just fail simple mathematics.

- **Look into the chain of thought of the model in the response. Do you observe any discrepancy between the bug prediction result and the model's reasoning? For example, the model's reasoning may mention the bug, but its prediction is "Correct." Similarly, the model may predict the code is "Buggy" without any indicator that it has reasoned about the program specification (natural language specification or tests)**
We observe discrepancies between the bug prediction result and the model's chain of thought reasoning for both vanilla and crated prompting. The discrepancies can be clearly observed in Table 8 and Table 9. Among the discrepancies for crafted prompting, except for the invalid predictions, the model predicts that the code is buggy but reasons that the code is correct for two tasks; and the model predicts that the code is correct but reasons that the code is buggy for one task.

Table 8: Bug Prediction Result and Model's Reasoning for Vanilla Prompting

HumanEval Task	Prediction	Reasoning
HumanEval/24	Buggy	Buggy
HumanEval/87	Invalid	Correct
HumanEval/113	Correct	Correct
HumanEval/108	Correct	Correct
HumanEval/65	Correct	Correct
HumanEval/136	Correct	Correct
HumanEval/112	Correct	Correct
HumanEval/8	Correct	Correct
HumanEval/38	Correct	Correct
HumanEval/153	Invalid[1]	Correct
HumanEval/154	Correct	Correct
HumanEval/36	Correct	Correct
HumanEval/93	Correct	Correct
HumanEval/103	Invalid	Correct
HumanEval/18	Correct	Correct
HumanEval/62	Correct	Correct
HumanEval/137	Correct	Correct
HumanEval/92	Correct	Correct
HumanEval/45	Correct	Correct
HumanEval/56	Invalid	Correct

[1]Invalid prediction means the model doesn't give a explicit prediction, or the prediction result is not enclosed by <start> <end>.

Table 9: Bug Prediction Result and Model's Reasoning for Crafted Prompting

HumanEval Task	Prediction	Reasoning
HumanEval/24	Invalid	Buggy
HumanEval/87	Buggy	Buggy
HumanEval/113	Buggy	Buggy
HumanEval/108	Buggy	Buggy
HumanEval/65	Buggy	Correct
HumanEval/136	Buggy	Buggy
HumanEval/112	Invalid	Correct
HumanEval/8	Correct	Correct
HumanEval/38	Buggy	Buggy
HumanEval/153	Buggy	Buggy
HumanEval/154	Buggy	Buggy
HumanEval/36	Buggy	Buggy
HumanEval/93	Correct	Correct
HumanEval/103	Buggy	Buggy
HumanEval/18	Buggy	Buggy
HumanEval/62	Correct	Correct
HumanEval/137	Buggy	Correct
HumanEval/92	Buggy	Buggy
HumanEval/45	Correct	Buggy
HumanEval/56	Correct	Correct