Experimenting Prompting Strategies across Different LLM Families

-Aryan Ghosh

GitHub Repository

Table of Contents

Part 1: Prompt Design & Code Generation

1. **HumanEval/155**

Base prompt:

def even_odd_count(num):

Given an integer, return a tuple that has the number of even and odd digits respectively.

Example:

even_odd_count(-12) ==> (1, 1)

even_odd_count(123) ==> (1, 2)

"""

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, ignore the sign of the number so only digits are evaluated. Then, iterate through each digit, count how many are even and how many are odd. Finally, return these counts as a tuple.

Prompt 2: Self-Planning

Before writing any code, outline the steps to solve the problem: 1) Take the absolute value of the number. 2) Convert the number to a string. 3) For each digit, check whether it is even or odd. 4) Keep count of even and odd digits. 5) Return the result as a tuple (even_count, odd_count). Now write the function following these steps.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

2. **HumanEval/101**

Base prompt:

```
def words_string(s):
    """
    You will be given a string of words separated by commas or spaces. Your task is to split the string into words and return an array of the words.
```

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, examine the string to see where commas and spaces separate words. Replace any commas with spaces so all separators are uniform. Then, split the string by spaces and return the resulting list after removing any empty entries.

Prompt 2: Self-Planning

Before writing code, outline the steps to solve the problem: 1) If the string is empty, return an empty list. 2) Replace all commas with spaces for consistency. 3) Split the string by spaces. 4) Remove any empty strings from the list. 5) Return the final list of words.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

3. **HumanEval/132**

Base prompt:

```
def is_nested(string):
    """Create a function that takes a string as input which contains only square brackets.
    The function should return True if and only if there is a valid subsequence of brackets where at
least one bracket in the subsequence is nested.
```

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, scan through the string to identify pairs of opening and closing brackets. Keep track of nested bracket patterns by looking for opening brackets followed by inner brackets before a closing bracket. If there is any valid nested structure and all brackets are balanced, return True; otherwise, return False.

Prompt 2: Self-Planning

Before writing code, outline the steps to solve the problem: 1) Check if the string is empty; if so, return False. 2) Use a stack to track brackets and nesting depth. 3) As you scan the string, increment depth for each '[' and decrement for each ']'. 4) If at any point depth exceeds 1, that means there is nesting—record that. 5) If the string is well-formed (all brackets match) and there was nesting, return True; else, return False.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Failed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

4. **HumanEval/136**

Base prompt:

def largest_smallest_integers(lst):

Create a function that returns a tuple (a, b), where 'a' is the largest of negative integers, and 'b' is the smallest of positive integers in a list.

If there is no negative or positive integers, return them as None.

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, iterate through the list to separate negative and positive numbers. Then, find the largest value among the negative numbers, and the smallest value among positive numbers. If either group is missing, return None for that value. Return a tuple (largest_negative, smallest_positive).

Prompt 2: Self-Planning

Before writing code, outline the steps required: 1) Check if the list is empty. 2) Create separate lists for negative and positive integers. 3) For negative integers, find the maximum value or None if the list is empty. 4) For positive integers, find the minimum value or None if the list is empty. 5) Return a tuple of these two values.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

5. **HumanEval/156**

Base prompt:

def int_to_mini_roman(number):

Given a positive integer, obtain its roman numeral equivalent as a string, and return it in lowercase.

Restrictions: 1 <= num <= 1000

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, create lists of the integer values and their corresponding Roman numeral strings. Then, starting from the largest value, repeatedly subtract from the input number, appending its Roman symbol to the result string until the number reaches zero. Finally, return the result in lowercase.

Prompt 2: Self-Planning

Before writing code, outline the steps to solve the problem: 1) Prepare lists of integer values and Roman numeral strings. 2) Iterate from largest to smallest value; for each value, determine how many times it fits into the number, and append the appropriate Roman symbol that many times. 3) Subtract the corresponding value until the number is reduced to zero. 4) Return the resulting Roman numeral string in lowercase.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

6. **HumanEval/160**

Base prompt:

def do_algebra(operator, operand): Given two lists operator and operand. The first list has basic algebra operations, and the second list is a list of integers. Use the two given lists to build the algebraic expression and return the evaluation of this expression.

Example:

operator=['+', '*', '-']

array=[2, 3, 4, 5]

result = 2 + 3 * 4 – 5 => result = 9

Note: The length of operator list is equal to the length of operand list minus one. Operand is a list of non-negative integers. Operator list has at least one operator, and operand list has at least two operands.

Prompt 1: Chain-of-Thought (CoT)

Think step by step: Start by constructing an expression string with the first operand. For each operator and the corresponding next operand, append them to the string to form the correct infix expression. After building the expression, evaluate it and return the result.

Prompt 2: Self-Planning

Before writing code, outline steps: 1) Initialize the expression with the first operand. 2) For each operator and subsequent operand, append both to the expression in sequence. 3) After constructing the full expression, evaluate it safely and return the result.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

7. **HumanEval/159**

Base prompt:

def eat(number, need, remaining):

You're a hungry rabbit, and you already have eaten a certain number of carrots, but now you need to eat more carrots to complete the day's meals.

You should return an array of [total number of eaten carrots after your meals, the number of carrots left after your meals]

If there are not enough remaining carrots, you will eat all remaining carrots, but will still be hungry.

Example:

eat(5, 6, 10) -> [11, 4]

eat(4, 8, 9) -> [12, 1]

eat(1, 10, 10) -> [11, 0]

eat(2, 11, 5) -> [7, 0]

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, check if the need can be satisfied with the remaining carrots. If there are enough carrots, eat only what is needed and update total and remaining. If not, eat all remaining carrots and return total eaten and 0 remaining.

Prompt 2: Self-Planning

Before writing code, outline the steps: 1) If the remaining carrots are greater than or equal to need, add 'need' to 'number' and subtract 'need' from 'remaining'. 2) If not enough carrots, add all 'remaining' to 'number' and set 'remaining' to 0. 3) Return both values in a list.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Failed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

8. **HumanEval/139**

Base prompt:

def special_factorial(n):

The Brazilian factorial is defined as:

brazilian_factorial(n) = n! * (n-1)! * (n-2)! * ... * 1!

where n > 0

For example:

>>> special_factorial(4) = 288

The function will receive an integer as input and should return the special factorial of this integer.

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, for each integer from 1 to n, calculate its factorial. Then multiply all these factorial values together to get the result. Return this final value.

Prompt 2: Self-Planning

Before writing code, outline the steps: 1) For each integer from 1 to n, compute its factorial. 2) Store all the factorials in a list or accumulate their product directly. 3) After calculating all factorials, multiply them together to get the final result. 4) Return the result.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

9. **HumanEval/141**

Base prompt:

def file_name_check(file_name):Create a function which takes a string representing a file's name, and returns 'Yes' if the file's name is valid, and returns 'No' otherwise.

A file's name is considered to be valid if and only if all the following conditions are met:

- There should not be more than three digits ('0'-'9') in the file's name.

- The file's name contains exactly one dot '.'.

- The substring before the dot should not be empty, and it starts with a letter from the latin alphabet ('a'-'z' and 'A'-'Z').

- The substring after the dot should be one of these: ['txt', 'exe', 'dll']

Examples:

file_name_check("example.txt") # => 'Yes'

file_name_check("1example.dll") # => 'No'

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, check if the file name contains exactly one dot. Next, split the name at the dot and validate both parts: check the extension is correct and the part before the dot starts with a letter. Count the digits in the name and ensure there aren't more than three. If all conditions pass, return 'Yes'; otherwise, return 'No'.

Prompt 2: Self-Planning

Before writing code, outline the steps: 1) Check the file name contains exactly one dot. 2) Split into name and extension; check extension against valid options and ensure name before the dot starts with a letter and isn't empty. 3) Count digits in the file name; fail if more than three. 4) If all criteria pass, return 'Yes'; otherwise, return 'No'.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

10. HumanEval/151

Base prompt:

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.

double_the_difference([1, 3, 2, 0]) == 1 + 9 + 0 + 0 = 10

double_the_difference([-1, -2, 0]) == 0

double_the_difference([9, -2]) == 81

double_the_difference([0]) == 0

If the input list is empty, return 0.

Prompt 1: Chain-of-Thought (CoT)

Think step by step: First, filter the list for positive odd integers only. Then, square each of those values and sum the results. Ignore any element that is negative, zero, or not an integer. If the input list is empty, return 0.

Prompt 2: Self-Planning

Before writing code, outline the steps: 1) Check if the input list is empty; if so, return 0. 2) Create a new list from elements that are both positive and odd integers. 3) Square each element in this filtered list. 4) Sum the squared values and return the result.

Strategy	**Prompt**	**LLM Used**	**Code Output Ref**	**Result (pass@1)**
CoT	Prompt 1	GPT5	GPT5-Prompt_1.py	Passed
CoT	Prompt 1	Perplexity	Perplexity-Prompt_1.py	Passed
Self-Planning	Prompt 2	GPT5	GPT5-Prompt_2.py	Passed
Self-Planning	Prompt 2	Perplexity	Perplexity-Prompt_2.py	Passed

Part 2: Debugging & Iterative Improvement

1. HumanEval/132: GPT5-Prompt 1(CoT)

- Test cases failure: I ran the humanEval test cases on the generated code and found that it failed on cases where nested brackets should be detected (e.g. '[[]]' , '[[[]]]', '[[][]]['), or falsely reported non-nested structures as nested.
    - Example failure: The code printed False when True was expected for '[[]]'.
- Changes made:
    - Initial code: The model used a depth counter but failed to define nesting properly, sometimes returning True for non-nested strings or False for valid ones.
    - Improved code: Added clarity about maximum depth and emphasized edge cases in the prompt helped the model return correct logic for nesting patterns. The improved code, after rerunning with the debugging prompt, passed the target test cases.
- The initial failure was probably due to a reasoning gap. The model tracked bracket balancing but did not distinguish between simple pairs (e.g. '[]') and actual nesting (e.g. '[[]]'). The code confused valid bracket pairs and required subsequence nesting.
    - Poor error handling: The model didn't account for all possible edge cases, especially those with only single brackets, empty strings, or misordered brackets.
- Difference in Families:
    - The initial Chain-of-Thought (CoT) prompt led to code that incorrectly equated balanced brackets with nested brackets, failing to detect actual nesting in strings. GPT5 required explicit clarification about depth computation and edge cases before it produced a correct solution.
    - Perplexity's code passed the test suite on the first attempt using the same CoT prompt. Its generated code already included logic for tracking bracket depth and correctly identified nesting, even for tricky edge cases. However, after the additional debugging, it showed a stronger initial understanding of the distinction between balanced and nested structures and still passed all tests.
- Original Prompt:

  def is_nested(string):

  Create a function that takes a string as input, which contains only square brackets.

  The function should return True if and only if there is a valid subsequence of brackets where at least one bracket in the subsequence is nested.

  Think step by step: First, scan through the string to identify pairs of opening and closing brackets. Keep track of nested bracket patterns by looking for opening brackets followed by inner brackets before a closing bracket. If there is any valid nested structure and all brackets are balanced, return True; otherwise, return False.
- Refined Prompt: Please debug for cases where the code may only count balanced brackets but does not check for actual nesting.

2. HumanEval/159: GPT5-Prompt 1(CoT)

- Test cases failure: I ran the HumanEval test cases on the generated code and found that it failed to handle certain cases correctly. The code produced outputs that did not match the expected results for various examples, including when not enough carrots remain to satisfy the need or when updating remaining carrots after eating.
    - Example failure: The code printed incorrect values for the test case eat (2, 11,5) , which was expected, revealing logic errors in both total calculation and leftovers.
- Changes made:
    - Initial code: The generated function tried to return total, left by using total = number + need and left = max (0, remaining - need), without considering the constraint that if there are not enough carrots left, the rabbit should eat all remaining carrots. This led to incorrect totals and leftovers when need > remaining.
    - Improved code: The revised prompt clarified: If remaining carrots are less than the need, eat all remaining carrots and set leftovers to zero; otherwise, eat only what is needed and subtract need from remaining. This modified the logic to:

$$if\ need\ <=\ remaining:$$

$$return\ [number\ +\ need, remaining\ -\ need]$$
$$else:$$
$$return\ [number\ +\ remaining, 0]$$

- The initial failure was probably due to:
    - Reasoning gap: The model focused on the formula: number + need, ignoring the dependency between need and remaining. It did not branch logic correctly for the case where not enough carrots were present for the need.
    - Poor error handling: There was no special handling for shortage scenarios, the model always subtracted need, possibly making leftovers negative or failing to satisfy the expected hungry rabbit outcome.
- Difference in Families:
    - GPT5 (CoT): Needed prompt clarification about handling the "not enough carrots" branch. The improved instructions resulted in a correct solution after debugging.
    - Perplexity: Passed both the initial and improved test suites without needing debugging; it correctly implemented the conditional branch to handle the carrot shortage up front and generated valid results for all edge cases.

- Original Prompt:
  def eat(number, need, remaining):
  You're a hungry rabbit, and you already have already eaten a certain number of carrots,but now you need to eat more carrots to complete the day's meals. You should return an array of [total number of eaten carrots after your meals, the number of carrots left after your meals]
  If there are not enough remaining carrots, you will eat all remaining carrots, but will still be hungry.
  Example:
  eat(5, 6, 10) -> [11, 4]
  eat(4, 8, 9) -> [12, 1]
  eat(1, 10, 10) -> [11, 0]
  eat(2, 11, 5) -> [7, 0]
  Think step by step: First, check if the need can be satisfied with the remaining carrots. If there are enough carrots, eat only what is needed and update total and remaining. If not, eat all remaining carrots and return total eaten and 0 remaining.
- Refined Prompt:
  If remaining carrots < need, eat all remaining carrots and set leftovers to zero. Otherwise, eat only what is needed and subtract need from remaining.
  For each case, return [total eaten, carrots left after meal].
  Ensure you handle edge cases where remaining is zero, need is zero, or number is zero.

Part 3: Innovation

- **Proposed Strategy:**
  Self-Reflect & Edit: After the LLM generates an initial solution, prompt the model to review its own code, identify mistakes or improvements, and output a revised version. This goes beyond standard Chain-of-Thought or Self-Planning by explicitly asking the model to act as its own code reviewer and editor before finalizing output.

- **Workflow:**
  1. Provide the standard task description and an initial code-generation prompt.
  2. Take the code generated by the LLM and feed it back to the model along with an explicit "self-edit"
  3. Test the revised code against similar or new edge cases.
  4. Record results and analyze effectiveness after this refined prompt.

- **Outcome of Strategy:**
  - The initial code from both GPT5 and Perplexity already passed all required test cases before the self-editing prompt.
  - After sending the self-review prompt, both models reviewed their logic, walked through reasoning for edge cases, and confirmed the original solution was correct.
  - Neither model proposed any substantive changes, but both gave justification for why the code was acceptable, noting the handling of edge cases and confirming correct outputs.

  Results Summary

LLM	Initial Pass	Self-Edit Pass	Improved Code	pass@k
GPT5	Yes	Yes	No (already optimal)	2
Perplexity	Yes	Yes	No (already optimal)	2

- **Analysis:**
  While the strategy did not improve results, it confirmed the correctness and robustness of existing code from both models. The self-edit phase served as a useful verification step but did not find bugs or yield improvements in this instance.

  This result demonstrates the strategy's limitation that if the model produces optimal code initially, self-edit serves to justify and check logic, but cannot improve what is already correct. The underlying assumption that multi-pass LLM review will improve quality will be best tested on error-prone or complex tasks that have a high number of edge cases.
  For problems that are already easy for modern LLMs, this approach primarily adds explainability and confidence.

GPT5 and Perplexity both reviewed the initial generated code well, while Perplexity's output included more comprehensive walkthroughs of inputs and edge case analysis. GPT5 not only confirmed the correctness of its original logic but also suggested an optional enhancement
GPT5 suggested that we should validate input arguments to ensure they are all non-negative integers. While this safeguard was not required, given the problem constraints, GPT5 recommended adding an input guard that would raise a ValueError if any arguments were negative or not integers

- **Initial Prompt:** [GPT5_Initial, Perplexity_Initial]
  def eat(number, need, remaining):
  You're a hungry rabbit, and you already have eaten a certain number of carrots, but now you need to eat more carrots to complete the day's meals.
  You should return an array of [total number of eaten carrots after your meals, the number of carrots left after your meals]
  If there are not enough remaining carrots, you will eat all remaining carrots, but will still be hungry.
  Example:
  eat(5, 6, 10) -> [11, 4]
  eat(4, 8, 9) -> [12, 1]
  eat(1, 10, 10) -> [11, 0]
  eat(2, 11, 5) -> [7, 0]
  Think step by step: First, check if the need can be satisfied with the remaining carrots. If there are enough carrots, eat only what is needed and update total and remaining. If not, eat all remaining carrots and return total eaten and 0 remaining.

- **Refined Prompt:** [GPT_refined, Perplexity_refined]
  Here is the code you generated for the rabbit carrot problem:

  [Initial Generated Code]

  Review this code for any possible logic errors, missed edge cases, or incorrect outputs. Check that:
  - The function correctly handles both cases where remaining carrots are enough and not enough to meet need.
  - The outputs match all given examples.
  - Edge cases like need = 0, remaining = 0, number = 0 are handled.
  If you find any issues, explain what is wrong and provide corrected code and reasoning. If the code is already correct, explain why.