GenAI for Software Development: Assignment 3

Comfort Ohajunwa
ccohajunwa@wm.edu

1 Introduction

Prompt engineering is the practice of crafting inputs to guide a large language model (LLM) toward generating a desired response. In this project, I explore the use of prompt engineering on 22 different code generation and understanding tasks. Specifically, I employ zero-shot, few-shot, self-consistency, prompt chaining, and chain-of-thought prompting strategies. For each task, I apply two different prompt strategies, which I input into two LLMs: Codestral 25.01 and GPT-4o-mini. Across all tasks, the models were set with a temperature of 0.7 and a token limit of 1024. For the self-consistency prompts, I have each model run the same prompt three times, generating three outputs each. In addition, I perform a comparative analysis by evaluating responses across models and prompting strategies. I provide both a quantitative analysis (using BLEU-4 and cosine similarity with CodeBERT embeddings) and a qualitative analysis based on my observations.

2 Task Analysis

For this project, I wrote scripts to automate much of the process, including reading the prompts from text files, inputting them into the two models, and generating responses. The scripts, prompts, and generated responses can be found at `https://github.com/cohajunwa/Prompt-Engineering`. In each of the subsections under Individual Analysis, I present a table with the task, prompt strategy, and evaluation scores for BLEU-4 and cosine similarity using CodeBERT embeddings (rounded to the nearest hundredth). I excluded the models' responses from the tables as there are 44 of them. Instead, the reader can visit the repository to see the model responses. The response data are stored in JSON files, which contain the task name, strategy, and outputs. Each task has a corresponding JSON file titled `[task #]_[strategy code]_responses.json`. The strategy codes are as follows: zs (zero-shot), fs (few-shot), sc (self-consistency), pc (prompt chaining), and cot (chain-of-thought). In addition, the outputs are also stored in text files titled `[task #]_[strategy code]_[model name]_[response #]_response.txt`.

2.1 General Observations

The wording of the outputs is very similar between Codestral and GPT-4o, indicating that both models were likely trained on similar datasets. However, Codestral often provided more detailed, broken-down explanations, whereas GPT-4o's responses tended to be more succinct. Their implementations were generally semantically identical, though one model would follow certain coding conventions (e.g., more idiomatic style, better modularity) better than the other.
As for prompting strategies, the models were generally able to produce comprehensive outputs with zero-shot. Few-shot, however, is particularly useful for refining how the model should respond. GPT-4o appeared slightly better at modeling its output after the provided examples than Codestral.

If the models are provided the same prompt multiple times, they provide nearly consistent results, though GPT-4o often had more variation between responses than Codestral. Prompt chaining may not be necessary for bug fixes (i.e., identify the bug, then fix), as the models generally seem "eager" to provide correct implementations even when explicitly asked to. Finally, when using the chain-of-thought strategy, both models tend to provide similar reasoning steps. GPT-4o seems more interested in code logic and implementation, whereas Codestral tends to provide more broken-down, beginner-friendly steps.

2.2 Individual Analysis

Task 1: Check if a Number is Prime (Python)

Goal: Code Summarization (Java)

Prompt Strategy: Zero-Shot

Prompt

Summarize the functionality of the following method:

```
public Map <String, Integer> countWordFrequency(List <String> words) {
        Map <String, Integer> freqMap = new HashMap <>() ;
        for (String word : words) {
                freqMap.put (word, freqMap.getOrDefault(word, 0) + 1);
        }
        return freqMap;
}
```

Evaluation Scores for Response

BLEU Score: 10.12

Cosine Similarity Score: 0.96

Both models provided accurate summaries of the functionality, though GPT-4o was more brief. Codestral, on the other hand, broke the functionality down into enumerated steps.

Goal: Code Summarization (Java)

Prompt Strategy: Self-Consistency

Prompt

Summarize the functionality of the following method:

```
public Map <String, Integer> countWordFrequency(List <String> words) {
        Map <String, Integer> freqMap = new HashMap <>() ;
        for (String word : words) {
                freqMap.put (word, freqMap.getOrDefault(word, 0) + 1);
        }
        return freqMap;
}
```

Evaluation Scores for Response 1

BLEU Score: 18.28

Cosine Similarity Score: 0.97

Evaluation Scores for Response 2

BLEU Score: 13.47

Cosine Similarity Score: 0.96

Evaluation Scores for Response 3

BLEU Score: 13.65

Cosine Similarity Score: 0.96

The models were generally consistent across all three instances, with subtle changes to wording and organization (e.g., Codestral enumerated three steps in its code summary for one instance and four in another).

Task 2: Bug Fixing (Python – Off-by-One)

Goal: Bug Fixing (Python – Off-by-One)
Prompt Strategy: Chain-of-Thought
Prompt

Identify and fix the off-by-one error in this function. Do it step-by-step:
 	
def sum_range(start, end):
  total = 0
  for i in range(start , end):
  total += i
  return total

Evaluation Scores for Response
BLEU Score: 40.66
Cosine Similarity Score: 1.00

Both models provided correct implementations and used similar steps, such as "understand the current behavior" and "identify/determine the intended/desired behavior." They also provided example test cases, even though testing was not mentioned in the prompt.

Goal: Bug Fixing (Python – Off-by-One)
Prompt Strategy: Prompt Chaining
Prompt 1

Identify the off-by-one error in this function:
def sum_range(start, end):
  total = 0
  for i in range(start , end):
  total += i
  return total

Evaluation Scores for Response 1
BLEU Score: 31.33
Cosine Similarity Score: 0.98
Prompt 2
Fix the error identified in the previous function.
Evaluation Scores for Response 2
BLEU Score: 68.50
Cosine Similarity Score: 1.00

Both models were able to identify the bug after the first prompt. Codestral's explanation was slightly more detailed, even outputting the specific line of the code that was wrong. After the second prompt, both models gave identical implementations.

Task 3: Bug Classification (C++)

Goal: Bug Classification (C++)
Prompt Strategy: Zero-Shot
Prompt

Classify the type of bug in the following C++ function:
 	
int * getArray(int size){
  int arr [size]; // Warning: local array
  return arr; // Bug: returning pointer to local variable
}

Evaluation Scores for Response
BLEU Score: 23.68
Cosine Similarity Score: 1.00

Both models identified the bug as a "dangling pointer" and gave a similar explanation. Codestral, however, also provided the correct implementation and driver code for testing it, even though the prompt did not ask the model to fix the code.

Goal: Bug Classification (C++)
Prompt Strategy: Self-Consistency
Prompt

Classify the type of bug in the following C++ function:
 	
int * getArray(int size){
  int arr [size]; // Warning: local array
  return arr; // Bug: returning pointer to local variable
}

Evaluation Scores for Response 1
BLEU Score: 10.84
Cosine Similarity Score: 0.98
Evaluation Scores for Response 2
BLEU Score: 19.20
Cosine Similarity Score: 0.99
Evaluation Scores for Response 3
BLEU Score: 11.23
Cosine Similarity Score: 0.99

The models were consistent in their responses – GPT-4o's responses were concise, while Codestral provided more detail and included a fixed implementation. Some of Codestral's responses even pointed out problematic lines of code. Only one of GPT-4o's responses included a fixed implementation.

Task 4: Generating Email Validators (Python + Regex)

Goal: Generating Email Validators (Python + Regex)
Prompt Strategy: Chain-of-Thought
Prompt

Complete the function using regex to validate basic email addresses. Do it step-by-step:
 	
def is_valid_email (email):
  # TODO: Complete using regex
  pass

Evaluation Scores for Response
BLEU Score: 42.47
Cosine Similarity Score: 1.00

Both models provided similar reasoning, though Codestral broke it down more than GPT-4o. For instance, Codestral's first step was to "Understand the structure of a basic email address." GPT-4o went straight into code implementation, with its first step being to "import the `re` module." Both models came up with different regex patterns. In addition, Codestral opted to compile the regex pattern first before matching it, while GPT-4o simply used `re.match`.

Goal: Generating Email Validators (Python + Regex)
Prompt Strategy: Prompt Chaining
Prompt 1
Write a regex expression to validate basic email addresses.
Evaluation Scores for Response 1
BLEU Score: 48.84
Cosine Similarity Score: 1.00
Prompt 2

Given the regex expression, complete the following function to validate basic email addresses:
 	
def is_valid_email (email):
  # TODO: Complete using regex
  pass

Evaluation Scores for Response 2
BLEU Score: 38.04
Cosine Similarity Score: 1.00

After the first prompt, both models provided regex expressions with detailed explanations. Surprisingly, the regex expression GPT-4o provided was the same as Codestral's rather than the one provided earlier. In addition, both models provided code implementations for Python, even though the prompt never explicitly asked for an implementation or mentioned a coding language. For prompt two, the models adapted their implementations for the provided incomplete function. Both models used `re.match`, but GPT-4o's code was slightly more Pythonic. While Codestral used an if-else statement for validating the email address, GPT-4o used a single line of code (`re.match(pattern, email) is not None`).

Task 5: Generating Flask APIs (Python)

Goal: Generating Flask APIs (Python)
Prompt Strategy: Zero-Shot
Prompt

Create a '/greet/' endpoint that returns a JSON greeting:
 	
from flask import Flask, jsonify
 	
app = Flask (__name__)
@app.route ('/greet/<username>')
def greet (username):
  # TODO Return a JSON greeting
  pass

Evaluation Scores for Response
BLEU Score: 42.78
Cosine Similarity Score: 1.00

Both models provided identical implementations with minor differences. They also discussed testing the endpoint. GPT-4o was more brief and simply noted what the output should be. Codestral, however, gave a step-by-step explanation for running the app.

Goal: Generating Flask APIs (Python)

Prompt Strategy: Few-Shot

Prompt

```
Here are some examples of endpoints:
#Example 1:
from flask import Flask, jsonify

app = Flask (__name__)
@app.route("/")
def hello_world():
  return "<p>Hello, World!</p>"

#Example 2:
from flask import Flask, jsonify

app = Flask (__name__)
@app.route('/health')
def health():
  return 'loadserver is healthy\n', 200


Now, create a '/greet/' endpoint that returns a JSON greeting:

from flask import Flask, jsonify

app = Flask (__name__)
@app.route ('/greet/<username>')
def greet (username):
  # TODO  Return a JSON greeting
  pass
```

Evaluation Scores for Response

BLEU Score: 46.15

Cosine Similarity Score: 1.00

Both models correctly implemented the function, though they seemed to interpret the prompt slightly differently. GPT-4o appeared to "think" that the example endpoints in the prompt were part of the same application, so it included all of them in its output code. Codestral simply gave the desired endpoint.

Task 6: SQL Schema Design (SQL)

Goal: SQL Schema Design (SQL)

Prompt Strategy: Self-Consistency

Prompt

```
Write the schema for a review app with users, books, and reviews:
-- TODO: Design schema with appropriate keys and constraints
-- Tables: users(id, name), books (id, title), reviews (id, user_id, book_id, rating)
```

Evaluation Scores for Response 1

BLEU Score: 39.87

Cosine Similarity Score: 0.99

Evaluation Scores for Response 2

BLEU Score: 43.26

Cosine Similarity Score: 1.00

Evaluation Scores for Response 3

BLEU Score: 33.29

Cosine Similarity Score: 1.00

The models incorporated the desired keys and constraints, though they implemented them differently. For instance, Codestral consistently made the user and book ids serial primary keys, while GPT-4o did that for two of its responses (for the third, it made them auto-incrementing integers). The reviews table had the most variation. Codestral, in particular, chose to add another key called `review_text` in two instances. GPT-4o had more variation across all three instances, not just in how it created the schema, but also in how it described it. Each response was structured differently,

whereas Codestral had a consistent structure for all its responses.

Goal: SQL Schema Design (SQL)
Prompt Strategy: Zero-Shot
Prompt
Write the schema for a review app with users, books, and reviews: -- TODO: Design schema with appropriate keys and constraints -- Tables: users(id, name), books (id, title), reviews (id, user_id, book_id, rating)
Evaluation Scores for Response
BLEU Score: 29.21
Cosine Similarity Score: 0.99

Codestral's output was relatively consistent with the other responses it gave for self-consistency. However, as mentioned earlier, GPT-4o varied its responses more. In this case, it had a different structure from the previous three responses and added new keys to its schema: `author`, `published_date`, `created_at`, and `updated_at` for the books table and `review_text` for the reviews table.

Task 7: Null Dereference Detection (Java)

Goal: Null Dereference Detection (Java)
Prompt Strategy: Zero-Shot
Prompt

Identify any null dereference risk:
 	
public int getLength(String s) {
  return s.length(); // What if s is null?
}

Evaluation Scores for Response
BLEU Score: 42.90
Cosine Similarity Score: 1.00

Both models correctly identified the null dereference risk and provided corrected implementations (even though the prompt did not explicitly ask for an implementation). Codestral provided two variations of an implementation – one that returns 0 if the string is null and another that raises an exception. GPT-4o just gave one implementation (returning 0) and briefly mentioned raising an exception as an alternative.

Goal: Null Dereference Detection (Java)

Prompt Strategy: Few-Shot

Prompt

```
Consider the following examples of null dereference risks:
Example 1:
public int getFirstElement(int arr[]){
        return arr[0];
}

If arr is null when passed to the method, then we have a null dereference when we call arr[0].

Example 2:
public int addThirdElement(int arr1[], int arr2[]){
        return arr1[2] + arr2[2];
}

If either arr1 or arr2 is null when passed to the method, then we have a null dereference.

Now, consider the following method. Identify any null dereference risk:
public int getLength(String s) {
  return s.length();
}
```

Evaluation Scores for Response

BLEU Score: 39.65

Cosine Similarity Score: 0.99

Both models modeled their responses to somewhat match the examples. For instance, both provided the buggy code first and then a description of the null dereference risk below. They still gave a suggested implementation. Codestral opted to just give one that raises an exception if the string is null. GPT-4o's code returned 0 instead, though there is a comment about throwing an exception.

Task 8: CSV Parser Variants

Goal: CSV Parser Variants (Python)

Prompt Strategy: Chain-of-Thought

Prompt

```
Improve the parser to support quoted fields. Do it step-by-step:

def parse_csv_line(line):
  return line.split(',') # Incomplete: doesn't handle quoted fields
```

Evaluation Scores for Response

BLEU Score: 32.68

Cosine Similarity Score: 1.00

Both models took slightly different approaches to improving the parser. Codestral started by enumerating the logic before describing how to implement it, while GPT-4o went straight into code. In addition, Codestral broke down the implementation into five steps. GPT-4o used four steps, with steps three and four focusing on edge cases and testing. Both implementations were semantically similar, though Codestral's code was cleaner and more Pythonic. GPT-4o handled certain edge cases, such as escape characters.

Goal: CSV Parser Variants (Python)
Prompt Strategy: Prompt Chaining
Prompt 1
Identify why the following parser is not able to handle quoted fields: def parse_csv_line(line):   return line.split(',') # Incomplete: doesn't handle quoted fields
Evaluation Scores for Response 1
BLEU Score: 18.65
Cosine Similarity Score: 0.99
Prompt 2
Improve the parser to support quoted fields.
Evaluation Scores for Response 2
BLEU Score: 17.70
Cosine Similarity Score: 0.99

After the first prompt, both models provided a detailed response to the prompt. GPT-4o, however, went further and pointed out edge cases. Both provided a shorter implementation using the built-in `csv` module. Interestingly, for the second prompt, GPT-4o provided an implementation that resembles Codestral's code with chain-of-thought (with considerations for the edge cases it described). Codestral gave the same implementation it provided for the first prompt, as well as an alternative implementation for processing multiple lines.

Task 9: Data Class to API Conversion (Kotlin)

Goal: Data Class to API Conversion (Kotlin)
Prompt Strategy: Zero-Shot
Prompt
Convert the data class to a REST API using Ktor: data class Product(val id: Int, val name: String, val price: Double) // TODO: Create GET and POST endpoints using Ktor
Evaluation Scores for Response
BLEU Score: 44.37
Cosine Similarity Score: 1.00

Both models gave similar step-by-step instructions for creating an API. Their application implementation is similar, but GPT-4o's version is more decoupled and modularized. Codestral also hard-coded a list of products, whereas GPT-4o made a mutable list that users can modify. While Codestral's code appears simpler, GPT-4o follows good practices, which makes its code more structured and extensible.

Goal: Data Class to API Conversion (Kotlin)
Prompt Strategy: Self-Consistency
Prompt
Convert the data class to a REST API using Ktor: data class Product(val id: Int, val name: String, val price: Double) // TODO: Create GET and POST endpoints using Ktor
Evaluation Scores for Response 1
BLEU Score: 42.13
Cosine Similarity Score: 1.00
Evaluation Scores for Response 2
BLEU Score: 47.44
Cosine Similarity Score: 1.00
Evaluation Scores for Response 3
BLEU Score: 56.67
Cosine Similarity Score: 0.99

Codestral's responses were generally consistent, including its implementations. GPT-4o's implementations, however, varied. Interestingly, its code resembled Codestral's more across all three instances, and its logic was more coupled.

Task 10: Function Summarization (Python)

Goal: Function Summarization (Python)
Prompt Strategy: Chain-of-Thought
Prompt
Reason about the following function step-by-step. Then, provide a brief summary of the function: def reverse_words(sentence):   return ''. join(sentence.split()[:: -1])
Evaluation Scores for Response
BLEU Score: 38.50
Cosine Similarity Score: 0.99

Both models gave identical steps in their reasoning, though Codestral also included code segments in its explanation and a short example at the end.

Goal: Function Summarization (Python)

Prompt Strategy: Few-Shot

Prompt

Here's some summaries for simple functions:

```
Example 1:
#Code
def add(x, y):
        return x + y

Summary: This function adds two numbers together.

Example 2:
#Code
def index_of_element(str_list, target_string):
        if target_str in str_list:
                return str_list.index(target_str)
        else:
                return -1

Summary: Given a list of strings and a target string, this function returns the index of the first occurrence of the target string if it exists. Otherwise, it returns -1.

Now, consider the following function and provide a brief summary:
def reverse_words(sentence):
  return ''. join(sentence.split()[:: -1])
```

Evaluation Scores for Response

BLEU Score: 3.09

Cosine Similarity Score: 0.96

Both GPT-4o and Codestral modeled their responses after the examples provided in the prompt, though GPT-4o's was shorter. GPT-4o's response was one sentence, while Codestral included the original code and a slightly longer summary.

Task 11: Prompt from Code Comments (Python)

Goal: Prompt from Code Comments (Python)

Prompt Strategy: Few-Shot

Prompt

Here's some examples of code generated based on code comments

```
Example 1:
# This function checks if a number is even
def is_even(n):
        return n % 2 == 0

Example 2:
# This function checks if a number is divisible by 10
def is_multiple_of_ten(n):
        return n % 10 == 0

Given the following comment, generate the function:
# This function checks if a number is prime
```

Evaluation Scores for Response

BLEU Score: 3.09

Cosine Similarity Score: 0.97

GPT-4o modeled its response after the examples in the prompt better than Codestral. GPT-4o's response was simply the function, whereas Codestral included an explanation.

Goal: Prompt from Code Comments (Python)
Prompt Strategy: Chain-of-Thought
Prompt
Given the following comment, generate the function. Do it step-by-step: # This function checks if a number is prime
Evaluation Scores for Response
BLEU Score: 45.19
Cosine Similarity Score: 1.00

Both models had slightly different approaches in their reasoning. Codestral focuses on explaining how to write the function using beginner-friendly language. For instance, for its "Step 2" it explained how to define and name the function in Python. GPT-4o, on the other hand, focused on the logic first, discussing details such as the input, output, and edge cases. At Step 4, it gave the full implementation. Codestral's approach may be more ideal for someone new to Python, but a seasoned programmer may appreciate GPT-4o's response more.

Task 12: Fixing Factorial Bug (Python)

Goal: Fixing Factorial Bug (Python)
Prompt Strategy: Self-Consistency
Prompt

Fix the bug when the input is 0:
 	
def factorial(n):
  result = 1
  for i in range(1, n):
  result *= i
  return result

Evaluation Scores for Response 1
BLEU Score: 35.01
Cosine Similarity Score: 1.00
Evaluation Scores for Response 2
BLEU Score: 42.95
Cosine Similarity Score: 1.00
Evaluation Scores for Response 3
BLEU Score: 25.48
Cosine Similarity Score: 0.99

Both models consistently provided similar code corrections with some minor tweaks. For instance, both the models changed the range to be from 1 to n+1. However, in two of Codestral's outputs, it also accounts for the edge case where $n < 0$ by raising a ValueError. GPT-4o did the same thing for two of its outputs, while the third output simply treated $n = 0$ as a special case.

Goal: Fixing Factorial Bug (Python)
Prompt Strategy: Prompt-Chaining
Prompt 1

Identify the bug in the following function.
 	
def factorial(n):
  result = 1
  for i in range(1, n):
  result *= i
  return result

Evaluation Scores for Response 1
BLEU Score: 26.25
Cosine Similarity Score: 1.00
Prompt 2
Now, fix the bug.
Evaluation Scores for Response 2
BLEU Score: 28.87
Cosine Similarity Score: 1.00

For the first prompt, both models provided corrections even though the prompt did not explicitly ask them to. Both models correctly identified the bug, but GPT-4o also suggested handling cases where $n < 0$. For the second prompt, the models repeated the corrected code they suggested for the first prompt. In addition, Codestral provided test examples, while GPT-4o enumerated the changes it made.

Task 13: Linked List Node Deletion (C)

Goal: Linked List Node Deletion (C)
Prompt Strategy: Chain-of-Thought
Prompt

Implement node deletion by value step-by-step:
struct Node {
  int data;
  struct Node* next;
};
 	
void deleteNode (struct Node** head, int key) {
  // TODO: Implement node deletion
}

Evaluation Scores for Response
BLEU Score: 36.19
Cosine Similarity Score: 1.00

Both models gave a list of similar steps, including accounting for edge cases (empty linked list, deleting head node), traversing the linked list, adjusting pointers, and dealing with cases where the node is not found. Their implementations of `deleteNode` were identical, though GPT-4o also provided an additional method for adding nodes and driver code for testing the method.

Goal: Linked List Node Deletion (C)
Prompt Strategy: Self-Consistency
Prompt

Implement node deletion by value:
struct Node {
  int data;
  struct Node* next;
};
 	
void deleteNode (struct Node** head, int key) {
  // TODO: Implement node deletion
}

Evaluation Scores for Response 1
BLEU Score: 58.90
Cosine Similarity Score: 1.00
Evaluation Scores for Response 2
BLEU Score: 47.77
Cosine Similarity Score: 1.00
Evaluation Scores for Response 3
BLEU Score: 53.67
Cosine Similarity Score: 1.00

Both models provided relatively consistent implementations across their responses. Interestingly, unlike with the chain-of-thought prompt, they each consistently produced a more complete class, with added methods for adding nodes, printing the list, and driver code for testing.

Task 14: Recursive Function Completion (Python)

Goal: Recursive Function Completion (Python)
Prompt Strategy: Chain-of-Thought
Prompt

Complete the recursive function for Fibonnaci. Do it step-by-step:
def fibonacci(n):
  # TODO : Base cases and recursive call
  pass

Evaluation Scores for Response
BLEU Score: 30.22
Cosine Similarity Score: 0.99

Both models provided similar steps, implementations, and example test cases. Both models also noted that the function is not efficient for large n and recommended "using memoization or an iterative approach."

Goal: Recursive Function Completion (Python)
Prompt Strategy: Zero-Shot
Prompt

Complete the recursive function for Fibonnaci:
def fibonacci(n):
  # TODO : Base cases and recursive call
  pass

Evaluation Scores for Response
BLEU Score: 23.04
Cosine Similarity Score: 1.00

Again, both models provided similar implementations and noted that the function had an exponential time complexity. However, Codestral also provided an optimized implementation using memoization. In addition, Codestral had a slightly different base case. Rather than considering $n = 0$ as its base case, it used $n \leq 0$.

Task 15: Constructor Completion (Python)

Goal: Constructor Completion (Python)
Prompt Strategy: Zero-Shot
Prompt

Complete the class constructor:
class Person:
  def __init__(self):
  # TODO: Add name, age, and optional email
  pass

Evaluation Scores for Response
BLEU Score: 31.38
Cosine Similarity Score: 0.99

Both models provided the same implementations, though Codestral's explanation was slightly longer.

Goal: Constructor Completion (Python)
Prompt Strategy: Self-Consistency
Prompt

Complete the class constructor:
class Person:
  def __init__(self):
  # TODO: Add name, age, and optional email
  pass

Evaluation Scores for Response 1
BLEU Score: 37.48
Cosine Similarity Score: 0.99
Evaluation Scores for Response 2
BLEU Score: 43.84
Cosine Similarity Score: 1.00
Evaluation Scores for Response 3
BLEU Score: 42.94
Cosine Similarity Score: 1.00

Both models consistently provided the same implementations. However, GPT-4o seemed to have more variation between responses. Interestingly, unlike with zero-shot, GPT-4o also provided example usages for all three instances, even though the prompt used for zero-shot and self-consistency was identical. On the other hand, one of Codestral's outputs with self-consistency is nearly identical to the output it provided with zero-shot.

Task 16: Binary Search Completion (Java)

Goal: Binary Search Completion (Java)
Prompt Strategy: Chain-of-Thought
Prompt

Complete the binary search implementation step-by-step:
public int binarySearch(int [] arr, int target) {
  int left = 0, right = arr.length - 1;
  while (left <= right) {
  int mid = (left + right) / 2;
  // TODO: Compare and adjust bounds
  }
  return -1;
}

Evaluation Scores for Response
BLEU Score: 50.87
Cosine Similarity Score: 1.00

Both models provided similar steps and explanations, though Codestral gave slightly more detail. Codestral's output may be more helpful for a beginner, whereas GPT-4o's response was more succinct. Their implementations were also identical with minor differences (e.g., `target < arr[mid]` vs `arr[mid] < target`).

Goal: Binary Search Completion (Java)
Prompt Strategy: Prompt Chaining
Prompt 1

Identify what is missing in the following binary search implementation.
public int binarySearch(int [] arr, int target) {
  int left = 0, right = arr.length - 1;
  while (left <= right) {
  int mid = (left + right) / 2;
  }
  return -1;
}

Evaluation Scores for Response 1
BLEU Score: 42.73
Cosine Similarity Score: 1.00
Prompt 2
Complete the binary search implementation.
Evaluation Scores for Response 2
BLEU Score: 42.81
Cosine Similarity Score: 0.99

Both models correctly identified the missing components in the given binary search implementation for the first prompt. They also gave a completed version of the binary search implementation, even though the first prompt did not explicitly ask them to. For the second prompt, both gave the same implementation they provided for the first prompt. GPT-4o, however, created a class containing the complete `binarySearch` method and driver code for testing.

Task 17: Bug Fixing (C++)

Goal: Self-Consistency Bug Fixing (C++)

Prompt Strategy: Few-Shot

Prompt

Consider the following examples where there are inconsistencies between the function name and logic:

```
//Buggy Code
int sum(int x, int y){
        return x - y; // Logic contradictions function name
}

//Fix:
int sum(int x, int y){
        return x + y; // Fix
}

//Buggy Code:
int avg(int x, int y){
        return x + y; // Logic contradictions function name
}

//Fix:
double avg(int x, int y){
        return (x + y)/2; // Fix
}
```

Now, resolve the inconsistency between the function name and logic in the following function:

```
// Supposed to return true if x is even
bool isOdd (int x) {
return x % 2 == 0; // Logic contradicts function name
}
```

Evaluation Scores for Response

BLEU Score: 37.91

Cosine Similarity Score: 0.99

Both models correctly identified the inconsistency given the first prompt. They also provided two potential implementations (one with a revised return statement and one with a revised function name), even though they were not explicitly asked to. In the second prompt, they both restated their implementations. Codestral's response was slightly longer, but gave essentially the same information as GPT-4o's.

Goal: Self-Consistency Bug Fixing (C++)
Prompt Strategy: Prompt Chaining
Prompt 1

Consider the following function and identify where the inconsistency occurs between the function name and the logic
// Supposed to return true if x is even
bool isOdd (int x) {
  return x % 2 == 0;
}

Evaluation Scores for Response 1
BLEU Score: 34.88
Cosine Similarity Score: 1.00
Prompt 2
Now, resolve the inconsistency.
Evaluation Scores for Response 2
BLEU Score: 28.87
Cosine Similarity Score: 1.00

Using the few-shot prompting strategy gave the models more context, which resulted in more concise outputs. In the examples given to the model, the solution was to change the return statement. So, Codestral and GPT-4o gave only one correct implementation.

Task 18: Prompt Chaining: Bug Identification → Fix (JavaScript)

Goal: Prompt Chaining: Bug Identification → Fix (JavaScript)
Prompt Strategy: Prompt Chaining
Prompt 1

Identify the bug:
function isEven(n) {
  return n % 2; // Returns 1 or 0, not true / false
}

Evaluation Scores for Response 1
BLEU Score: 37.52
Cosine Similarity Score: 1.00
Prompt 2
Fix the bug.
Evaluation Scores for Response 2
BLEU Score: 36.87
Cosine Similarity Score: 1.00

Both models' responses were nearly identical, with minor differences in wording. After the first prompt, both models correctly identified the bug and provided corrected code, even though they were not asked for an implementation. For the second prompt, they simply restated their response to the first prompt.

Goal: Prompt Chaining: Bug Identification → Fix (JavaScript)

Prompt Strategy: Few-Shot

Prompt

```
Consider the following examples of bug fixes:
Example 1:
// Buggy Code
function multiply(n1, n2){
        return n1 + n2; // Bug: Returns the sum of two numbers instead of the product
}

// Fixed Code
function multiply(n1, n2){
        return n1 * n2;
}

Example 2:
// Buggy Code
function modulo(a, b){
        return a%b == 0; // Bug: Returns True/False instead of a mod b
}

// Fixed Code
function modulo(a, b){
        return a%b;
}

Now, consider the following function. Identify and fix the bug:
function isEven(n) {
  return n % 2; // Returns 1 or 0, not true / false
}
```

Evaluation Scores for Response

BLEU Score: 46.11

Cosine Similarity Score: 1.00

Both models' responses were similar to those generated with prompt chaining. Providing the models with examples did little to change their outputs.

Task 19: Summary Decomposition (C++)

Goal: Summary Decomposition (C++)

Prompt Strategy: Chain-of-Thought

Prompt

```
Decompose the high-level comment/summary into logical steps. Do it step-by-step:
// Function that validates an input, calculates square, and returns result
int process (int x) {
  if (x < 0) return -1; return x * x;
}
```

Evaluation Scores for Response 1

BLEU Score: 25.74

Cosine Similarity Score: 0.99

Interestingly, both models interpreted the prompt slightly differently. Both models outlined similar steps, including input validation, square calculation, and returning the result. However, GPT-4o did not go further than this, while Codestral also rewrote the function with comments detailing where each step was implemented. Then, it provided an additional explanation, where it wrote each step with the corresponding block of code.

Goal: Summary Decomposition (C++)
Prompt Strategy: Self-Consistency
Prompt

Decompose the high-level comment/summary into logical steps.
// Function that validates an input, calculates square, and returns result
int process (int x) {
  if (x < 0) return -1; return x * x;
}

Evaluation Scores for Response 1
BLEU Score: 48.07
Cosine Similarity Score: 0.99
Evaluation Scores for Response 2
BLEU Score: 38.88
Cosine Similarity Score: 1.00
Evaluation Scores for Response 3
BLEU Score: 34.18
Cosine Similarity Score: 1.00

Both models' responses were fairly consistent across all three prompts. Their responses were slightly more concise than the responses they gave for chain-of-thought.

Task 20: Purpose Inference → Completion (Python)

Goal: Purpose Inference → Completion (Python)
Prompt Strategy: Few-Shot
Prompt

Here are a few examples of function completion based on intent:
Example 1:
# Input: 	
def calculate_sum(scores):
  total = 0
  # TODO: Complete to return sum
  pass
 	
# Output: 	
def calculate_sum(scores):
  total = 0
  for score in scores:
  total += score
  return sum
 	
Example 2:
# Input: 	
def get_maximum(scores):
  # TODO: Return maximum score
 	
# Output: 	
def get_maximum(scores):
  return max(scores)
 	
Complete the function based on intent:
# Input: 	
def calculate_average(scores):
  total = 0 # TODO: Complete to return average
  pass

Evaluation Scores for Response
BLEU Score: 5.36
Cosine Similarity Score: 0.98

Both models gave similar implementations, though GPT-4o's implementation was cleaner and more Pythonic. For instance, while Codestral used a for-loop, GPT-4o used Python's built-in `sum` function. GPT-4o also considered an edge case where the list was empty, whereas Codestral did not. Additionally, GPT-4o modeled its response after the few-shot examples by simply responding with just the code. Codestral, however, included a brief description before and after the code it provided.

Goal: Purpose Inference → Completion (Python)
Prompt Strategy: Self-Consistency
Prompt

Complete the function based on intent:
def calculate_average(scores):
  total = 0 # TODO: Complete to return average
  pass

Evaluation Scores for Response 1
BLEU Score: 35.92
Cosine Similarity Score: 1.00
Evaluation Scores for Response 2
BLEU Score: 40.99
Cosine Similarity Score: 1.00
Evaluation Scores for Response 3
BLEU Score: 26.76
Cosine Similarity Score: 1.00

Prompting the models three times resulted in slightly different variations across the instances. Each model provided more detail regarding the function's intent than it did for few-shot. GPT-4o consistently used Python's built-in `sum` function for all three responses, whereas Codestral used it for only two instances. Both models also consistently considered the edge case where the list was empty.

Task 21: Full-File Bug Detections and Refactoring (Python)

Goal: Full-File Bug Detections and Refactoring (Python)
Prompt Strategy: Zero-Shot
Prompt

Analyze the following utility script. Identify any potential logic issues or design flaws. Then, refactor it for better readability, correctness, and safety.
 	
# utils.py - Script to parse and summarize numeric CSV files 	
 	
import csv
 	
def read_csv(filepath):
  with open(filepath , 'r') as f:
  return [row for row in csv.reader(f)]
def summarize_column(data, index):
  values = [float(row[index]) for row in data [1:]] # skip header
  total = sum(values)
  avg = total / len (values)
  return total, avg
def main():
  filepath = 'data .csv '
  data = read_csv(filepath)
  total, avg = summarize_column(data, 1)
  print("Total :", total)
  print("Average :", avg)
if __name__ == '__main__ ':
  main()

Evaluation Scores for Response
BLEU Score: 44.97
Cosine Similarity Score: 1.00

Both models identified similar errors and provided nearly identical implementations (with subtle

differences). The most visible difference, however, is the amount of documentation. Both models noted the lack of documentation in the original script and included docstrings in their implementations. While GPT-4o's docstrings were just one-line explanations, Codestral's docstrings were much more detailed, including the arguments, return values, and raised exceptions.

Goal: Full-File Bug Detections and Refactoring (Python)

Prompt Strategy: Self-Consistency

Prompt

Analyze the following utility script. Identify any potential logic issues or design flaws. Then, refactor it for better readability, correctness, and safety.

```
# utils.py - Script to parse and summarize numeric CSV files

import csv

def read_csv(filepath):
  with open(filepath , 'r') as f:
  return [row for row in csv.reader(f)]
def summarize_column(data, index):
  values = [float(row[index]) for row in data [1:]] # skip header
  total = sum(values)
  avg = total / len (values)
  return total, avg
def main():
  filepath = 'data .csv '
  data = read_csv(filepath)
  total, avg = summarize_column(data, 1)
  print("Total :", total)
  print("Average :", avg)
if __name__ == '__main__ ':
  main()
```

Evaluation Scores for Response 1

BLEU Score: 42.43

Cosine Similarity Score: 1.00

Evaluation Scores for Response 2

BLEU Score: 38.37

Cosine Similarity Score: 1.00

Evaluation Scores for Response 3

BLEU Score: 39.99

Cosine Similarity Score: 1.00

Both models provided consistently similar outputs across all three instances.

Task 22: Code Completion and Robustness Enhancement (Python)

Goal: Code Completion and Robustness Enhancement (Python)

Prompt Strategy: Chain-of-Thought

Prompt

Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly. Do it step-by-step:

```
# file_processor.py - Incomplete script for processing text files

import string

def load_file(filepath):
  with open(filepath, 'r') as f:
    return f.readlines()

def clean_line(line):
  # TODO: Remove punctuation and make lowercase
  pass

def count_words(lines):
  word_counts = {}
  for line in lines:
  clean = clean_line(line)
  for word in clean.split():
    word_counts[word] = word_counts.get(word, 0) + 1
  return word_counts

def main():
  filepath = 'input.txt'
  lines = load_file(filepath)
  counts = count_words(lines)
  for word, count in sorted(counts.items()):
    print (f"{ word }: { count }")

if __name__ == '__main__ ':
  main()
```

Evaluation Scores for Response

BLEU Score: 41.93

Cosine Similarity Score: 0.99

Both models provided identical steps, though they presented them slightly differently. For each step, Codestral explained how to implement it in the code. On the other hand, GPT-4o listed the steps and gave the full code implementation at the end.

Goal: Code Completion and Robustness Enhancement (Python)

Prompt Strategy: Prompt Chaining

Prompt 1

Identify what's wrong with the following script:

```
# file_processor.py - Incomplete script for processing text files

import string

def load_file(filepath):
  with open(filepath, 'r') as f:
    return f.readlines()

def clean_line(line):
  # TODO: Remove punctuation and make lowercase
  pass

def count_words(lines):
  word_counts = {}
  for line in lines:
  clean = clean_line(line)
  for word in clean.split():
    word_counts[word] = word_counts.get(word, 0) + 1
  return word_counts

def main():
  filepath = 'input.txt'
  lines = load_file(filepath)
  counts = count_words(lines)
  for word, count in sorted(counts.items()):
    print (f"{ word }: { count }")

if __name__ == '__main__ ':
  main()
```

Evaluation Scores for Response 1

BLEU Score: 61.99

Cosine Similarity Score: 1.00

Prompt 2

Now, complete the script.

Evaluation Scores for Response 2

BLEU Score: 41.94

Cosine Similarity Score: 1.00

After the first prompt, both models identified the errors in the original script and provided a corrected version, even though the prompt did not explicitly ask for one. Both models focused on completing the `clean_line` function, the indentation error in the `count_words` function, and the curly quotes in `if __name__ == '__main__'`. After the second prompt, however, both models included these corrections as well as some error handling in the `load_file` function and edge case handling in the `main` function.