GenAI for Software Development Assignment 3

Ella Bryant

egbryant01@wm.edu

1 Introduction

The Using Different LLMs repository sought to explore prompt engineering through analyzing the outputs of different prompting methods across different software development tasks and models. The source files can be found at

2 Implementation

There are 22 different software development tasks that are used to evaluate the effectiveness of prompting methods across different models. Each of these tasks is assigned two prompt strategies that were randomly distributed from a pool of prompting strategies: 8 Zero-Shot Prompting, 8 Few-Shot Prompting, 7 Chain-of-Thought Prompting, 7 Role-Playing Prompting, 7 Prompting Chaining, and 7 Self-Consistency Prompting. The outcome of this distribution is detailed below in the results section. Once each of these prompts was crafted for the specifications, the prompts were entered into the “Prompts.csv” and used to prompt each of the following models: Claude, ChatGPT, Gemini, and Llama. The outputs from each of these models are stored in “claude.csv”, “gpt.csv”, “gemini.csv”, and “llama.csv” respectively in the corresponding location to where the prompts are stored in the prompt file.

3 Results

1. Task: **Code Summarization (Java).** 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;

}

**Prompting Strategy 1: Chain-of-Thought Prompting**

Prompt: Summarize the functionality of the following method implemented in Java code providing step-by-step justification and reasoning to get the output.

Output: This Java method, countWordFrequency, takes a list of strings as input list of strings as input and returns a map where each unique string from the input list is a key, and its corresponding value is the number of times that string appears in the list.

\*\*step by step breakdown\*\*

In essence, the method iterates through the list of words. For each word, it checks if the word is already the frequency map. If it is the count for the word is incremented. If it’s not, the word is added to the map with a count of 1. Finally, the map containing the word frequencies is returned.

Analysis: The above output is from the Gemini model. This output was selected because it had the most coherent natural language summary out of the different models. The summary is fundamentally correct and shows an advanced understanding of the java syntax in the step-by-step breakdown, going through each line of code to further develop the summary.

**Prompting Strategy 2: Few-Shot Prompting**

Prompt: Code: ‘’’

public int countElements(List<T> list) {

if (list == null) {

return 0;

}

int count = 0;

for (<T> element : list) {

count = count + 1;

}

return count;

}

‘’’

Summary: The code counts all the elements in an ambiguously typed list by first checking that the list is not empty before iterating through each of the elements incrementing a tracker for each.

Code: ‘’’

public static <T> int countMatchingElements(List<T> list, Predicate<T> condition) {

if (list == null || condition == null) {

return 0;

}

int count = 0;

for (T element : list) {

if (condition.test(element)) {

count++;

}

}

return count;

}

‘’’

Summary: The code counts all the elements in a list of an ambiguous type that match a predetermined element and condition by first checking that the list is not empty and the condition is not empty before iterating through each of the elements incrementing a tracker for each time the condition is met.

Summarize the functionality of the following method implemented in Java code using the above examples as models for how to solve the task.

Output: The code calculates the frequency of each unique word in a list of strings by creating a HashMap to store word-frequency pairs, then iterating through each word in the list and either incrementing its existing count or adding it with a count of 1 if it’s encountered for the first time.

Analysis: The above output is from the Claude model. This output was selected because it was the most concise and descriptive out of the different models. The summary is fundamentally correct while not including as many details as assisted by the concise examples in the prompt.

**Model Comparison and Analysis**

Both prompts contextually have the same outputs across the prompting strategies and the models. However, the most effective strategy depends on your preferences with simplicity. Chain-of-Thought goes through each line of code to develop its understanding of the model, but this results in a convoluted and redundant explanation. All of the models except Gemini did not have a detailed natural language summary contained in their outputs focusing more on the step-by-step section of the prompt. In the Few-Shot Prompting, since the examples were concise, this seems to have contributed to all the models selecting concise natural language summaries of the code’s function. However, the Llama output for Few-Shot Prompting still had a step-by-step breakdown making the output convoluted.

1. Task: **Bug Fixing (Python – Off-by-One).** Identify and fix 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

**Prompting Strategy 1: Prompt Chaining**

Prompt 1: Identify the off-by-one error in the following Python function.

Prompt 2: Fix the bug from the Python function that was identified above, outputting the fixed code.

Output: ‘’’

def sum\_range(start, end):

total = 0

for i in range(start, end+1):

total += i

return total

‘’’

Analysis: The above output is from the GPT model. The output was selected because it identified the error and fixed the bug, giving an option to improve implementation. The fixed code is fundamentally correct and identifies the correct code. However, the prompt chaining may not have been effective as it jumped ahead without waiting for an explicit follow-up prompt.

**Prompting Strategy 2: Few-Shot Prompting**

Prompt: Code: ‘’’

def print\_natural\_numbers (n):

for i in range(0, n+1):

print(i)

‘’’

Identified Error and Fixed Code: The range function includes the first input in its range, making the first output 0 for each n.

‘’’

def print\_natural\_numbers (n):

for i in range(1, n+1):

print(i)

‘’’

Code: ‘’’

def sum\_slice(lst, start, end):

total = 0

for i in range(start, end):

total += arr[i]

return total

‘’’

Identified Error and Fixed Code: The range function excludes the second input in its range, including only start through end-1 in its summation.

‘’’

def sum\_slice(lst, start, end):

total = 0

for i in range(start, end+1)”

total += arr[i]

return total

‘’’

Identify and fix the off-by-one error in the following Python function using the above examples for models of how to do so and output the fixed code.

Output: Option 1:

‘’’

def sum\_range(start, end(:

total = 0

for I in range(start, end + 1):

total += i

return total

‘’’

Option 2:

‘’’

def sum\_range(start, end):

return sum(range(start, end + 1))

‘’’

Analysis: The above output is from the Llama model. The output was selected because the model also provided a second option for fixing the bug that is more efficient. The fixed code in both options is correct in both identifying and fixing the off-by-one error.

**Model Comparison and Analysis**

Both outputs contextually have the same outputs across the prompting strategies. Certain models were better at optimizing the code for efficiency than others with both GPT and Llama attempting to do so. In Prompt Chaining, Llama presented an incorrect optimized version. In Few-Shot Prompting, the models seemed less likely to attempt efficiency optimizations due to the examples not containing those. However, due to the nature of the task, Few-Shot Prompting appeared more effective than Prompt Chaining since Prompt Chaining had the models jumping ahead and fixing the code before explicitly being prompted to do so. Thus, due to the simplicity of the task, a more simple approach to prompting may be more efficient.

1. Task: **Bug Classification (C++).** 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

}

**Prompting Strategy 1: Few-Shot Prompting**

Prompt: Code: ‘’’

int add\_5(int a) {

double b = 5;

return a + b;

}

‘’’

Classified Bug: There is a type inconsistency between a being a int and b being a double. The addition will convert a to a double, but the output is also an int which must then convert the double a+b back to an int.

Code: ‘’’

Void create\_and\_delete() {

int\* ptr = new int(10);

std::cout << “Value: “ << \*ptr << std::endl;

delete ptr;

std::cout << “After delete, value: “ << \*ptr << std::endl;

}

‘’’

Classified Bug: There is dangling error because the pointer is called after the memory allocation it was pointing to got deleted.

Classify the type of bug in the following C++ function using the above examples for models of how to do so.

Output: Classified Bug: This contains a dangling pointer bug. The function declares a local array arr on the stack, which will be automatically deallocated when the function exits. Returning this pointer creates a dangling pointer because it references memory that is no longer valid once the function returns. When the caller attempts to access this memory through the returned pointer, it will result in undefined behavior since the memory has already been deallocated.

Analysis: The above output is from the Claude model. This output was selected because, while each model fundamentally returned the same meaning, this one had the easiest readability. The result correctly classifies the bug and describes it well.

**Prompting Strategy 2: Zero-Shot Prompting**

Prompt: Classify the type of bug in the following C++ function.

Output: The function getArray creates a local array arr of size size. However, returning a pointer to this local array is problematic because the array is allocated on the stack, and its memory will be deallocated once the function scope ends. Therefore, returning a pointer to this memory creates a dangling pointer, a pointer that points to memory that is no longer valid. This can lead to undefined behavior when the pointer is dereferenced outside the function.

\*\*fix\*\*

Analysis: The above output is from the GPT model. This output is selected because it correctly identifies “dangling pointer error” and offers a potential bug fix. The result is correct with additionally unspecified functionality.

**Model Comparison and Analysis**

Both prompts contextually have correctly generated outputs. However, the Few-Shot Prompting strategy was more likely to attribute it specifically to “dangling pointer error,” regardless the described error was as intended. The Zero-Shot Prompting strategy had the tendency to be more open ended in its description of the bug as the models didn’t have the examples to model an output from. The GPT model was the only model to attempt to fix the bug without being specifically prompted to do so.

1. Task: **Generating Email Validators (Python + Regex).** Complete the function using regex to validate basic email addresses.

**Prompting Strategy 1: Few-Shot Prompting**

def is\_valid\_email(email):

# TODO: Complete using regex

pass

Prompt: Code: ‘’’

def is\_valid\_phone\_numbers(number):

# TODO: Complete using regex

pass

‘’’

Completed Code: ‘’’

def is\_valid\_phone\_numbers(number):

phone\_pattern = r’\(?\d{3}\})?[-.\s]?\d{3}[-.\s]?\d{4}’

return re.match(phone\_pattern, number)

‘’’

Code: ‘’’

def is\_valid\_date(date):

# TODO: Complete using regex

pass

‘’’

Completed Code: ‘’’

def is\_valid\_date(date):

date\_pattern = r’\b(?:\d{2}[-/]\d{2}[/]\d{4}|\d{2}/\d{2}/\d{4})\b'

return re.match(date\_pattern, date)

‘’’

Complete the following Python function using Regex to validate basic email addresses using the above examples for models of how to do so.

Output: ‘’’

import re

def is\_valide\_email(email):

email\_pattern = r’^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$’

return re.match(email\_pattern, email)

‘’’

Explanation:

* ^[a-zA-Z0-9.\_%+-]+: Ensures the email starts with a valid username consisting of alphanumeric characters and special characters like ., \_, %, +, and -.
* @: The ""@"" symbol separating the username from the domain name.
* [a-zA-Z0-9.-]+: The domain name part, which can include alphanumeric characters, hyphens, and periods.
* \.: The dot (.) separating the domain and the top-level domain (TLD)
* [a-zA-Z]{2,}$: The TLD, which must consist of at least two alphabetic characters.

Analysis: The above output is from the GPT model. The output was selected because it was the most syntactically sound. This result was correct and had a coherent and understandable connection

**Prompting Strategy 2: Zero-Shot Prompting**

Prompt: Complete the following Python function using Regex to validate the basic email addresses.

Output: ‘’’

import re

def is\_valid\_email(email):

# Email regex pattern – matches username@domain.tld format

email\_pattern = r’[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9.]+$

# Check if the email matches the pattern

if re.match(email\_pattern, email):

return True

else:

return False

‘’’

You could simplify the return statement to just return bool(re.match(email\_pattern, email)) if you prefer a more concise approach.

Analysis: The above output is from the Claude model. The output was selected because it was syntactically correct. Despite being correct, the method is slightly redundant as the second section could have been done in a single line of code as the model points out. And there should have been a character minimum on the TLD section. However, the comments are considered better coding practice.

**Model Comparison and Analysis**

Models appear to struggle to tackle the syntax associated with implementing regex in Python. GPT was the only model to have correct syntax for both prompting strategies as Claude had an import statement embedded in a function block for the Few-Shot Prompting strategy. The Few-Shot Prompting was not as effective as the Zero-Shot Prompting because Few-Shot Prompting had more syntactical errors despite the added examples for implementation. The Zero-Shot Prompting strategy also had the added coding feature of proper commenting which is generally considered a better coding practice.

1. Task: **Generating Flask APIs (Python).** Create a ‘/greet/<username>’ endpoint that returns a JSON greeting.

**Prompting Strategy 1: Zero-Shot Prompting**

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

@app.route(‘/greet/<username>’)

def greet(username):

# TODO: Return a JSON greeting

pass

Prompt: Using the Flask API in Python, create a ‘/greet/<username>’ endpoint that returns a JSON greeting for the following Python code.

Output: ‘’’

import flask import Flask, jsonify

app = Flask(\_\_name\_\_)

@app.route(‘/greet/<username>’)

def greet(username):

greeting = {‘message’: f’Hello, {username}!’}

return jsonify(greeting)

if \_\_name\_\_ == ‘\_\_main\_\_’:

app.run(debug=True)

‘’’

Analysis: The above output is from the Llama model. This output was selected because it has syntactic correctness and contains a user-friendly response. The result is correct and contains an explanation of the code and instructions on how to test the code.

**Prompting Strategy 2: Self-Consistency Prompting**

Prompt: Using the Flask API in Python, create a ‘/greet/<username>’ endpoint that returns a JSON greeting for the following Python code.

The above prompt was passed 5 times.

Output: ‘’’

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

@app.route(‘/greet/<username>’)

def greet(username):

return jsonify({“message”: f”Hello, {username}!”,

“status”: “success”

})

if \_\_name\_\_ == ‘\_\_main\_\_’:

app.run(debug=True)

‘’’

Analysis: The above output is from the Claude model. This output was selected because it is syntactically correct. The other models had syntactic errors concerning double quotes; however, this result is correct and more concise.

**Model Comparison and Analysis**

Models appear to struggle with the syntax of double quotations in python. All of the models had an issue with this syntax in at least one of the prompting strategies. The results of both prompting strategies were relatively similar despite these errors with the Self-Consistency Prompting being slightly more concise. The Claude model had the additional output of “status”: “success” which was an additional detailing indicating good programming practices.

1. Task: **SQL Schema Design (SQL).** Write the schema for review app with users, books, and reviews.

**Prompting Strategy 1: Zero-Shot Prompting**

-- TODO: Design schema with appropriate keysand constraints

-- Tables: users(id, name), books(id, title), reviews(id, user\_id, book\_id,

rating)

Prompt: In the SQL language, write the schema for a review app with users, books and review based on the following base design.

Output: ‘’’

CREATE TABLE users (

id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

password\_hash VARCHAR(255) NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Create the books table

CREATE TABLE books (

id SERIAL PRIMARY KEY,

title VARCHAR(255) NOT NULL,

author VARCHAR(100) NOT NULL,

isbn VARCHAR(20) UNIQUE,

publication\_year INT,

genre VARCHAR(50),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

-- Create the reviews table

CREATE TABLE reviews (

id SERIAL PRIMARY KEY,

user\_id INT NOT NULL,

book\_id INT NOT NULL,

rating INT NOT NULL CHECK (rating BETWEEN 1 AND 5),

review\_text TEXT,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users(id) ON DELETE CASCADE,

FOREIGN KEY (book\_id) REFERENCES books(id) ON DELETE CASCADE,

UNIQUE (user\_id, book\_id) –- Ensure one review per user per book

);

-- Create indexes for performance

CREATE INDEX idx\_reviews\_user\_id ON reviews(user\_id);

CREATE INDEX idx\_reviews\_book\_id ON reviews(book\_id);

CREATE INDEX idx\_reviews\_books\_title ON books(title);

‘’’

Analysis: The above output is from the Claude model. This output was selected because it was the most robust implementation, can be run in MySQL, and lack syntactical errors. The result is correct and relatively robust despite the lack of detail given in the prompt.

**Prompting Strategy 2: Role-Playing Prompting**

Prompt: You are a software developer and expert in SQL Schema Design. You have been tasked with writing the schema for a review app with users, books, and reviews with following base design.

Write the schema design you come up with.

Output: ‘’’

-- Table for storing user information

CREATE TABLE users (

id INT PRIMARY KEY AUTO\_INCREMENT, -- User’s unique identifier

name VARCHAR(255) NOT NULL – User’s name, cannot be NULL

);

-- Table for storing book information

CREATE TABLE books (

id INT PRIMARY KEY AUTO\_INCREMENT, -- Book’s unique identifier

title VARCHAR(255) NOT NULL -- Book's title, cannot be NULL

);

-- Table for storing reviews with foreign key relationships

CREATE TABLE reviews (

id INT PRIMARY KEY AUTO\_INCREMENT, -- Review’s unique identifier

user\_id INT NOT NULL, -- Foreign key to the users table

book\_id INT NOT NULL, -- Foreign key to the books table

rating INT CHECK (rating >= 1 AND rating <= 5),

-- Rating between 1 and 5

review\_text TEXT, -- Optional review text

FOREIGN KEY (user\_id) REFERENCES users(id) ON DELETE CASCADE,

-- User reference, delete reviews if user is deleted

FOREIGN KEY (book\_id) REFERENCES books(id) ON DELETE CASCADE

-- Book reference, delete reviews if book is deleted

);

-- Index on user\_id and book\_id in reviews for better query performance

CREATE INDEX idx\_reviews\_user\_book ON reviews(user\_id, book\_id);

‘’’

Analysis: The above output is from the GPT model. This output was selected because it is implemented in MySQL. The result is correctly implemented with comments showing proper programing techniques.

**Model Comparison and Analysis**

Both prompts contextually have correctly generated outputs. The models were relatively uniform in their outputs except for Claude that delivered a more robust answer in the Zero-Shot Prompting strategy. And each of the models had an issue in portability or MySQL compatibility except for GPT. The Role-Playing Prompting strategy appeared to have better coding strategies than the Zero-Shot Prompting with proper commenting practices. However, the Zero-Shot Prompting strategy appeared to have more robust code than Role-Playing Prompting strategy.

1. Task: **Null Dereference Detection (Java).** Identify any null dereference risk.

public int getLength(String s) {

return s.length(); // What if s is null?

}

**Prompting Strategy 1: Self-Consistency Prompting**

Prompt: Identify any null dereference risk in the following Java code.

The above prompt was used 5 times.

Output: ‘’’

pubic int getLength(String s) {

if (s == null) {

return 0; // Or implement alternative handling

return s.length();

‘’’

Analysis: The above output is from the Gemini model. This output was selected ambiguously as all models fundamentally had the same output. The result is correct in its identification and fix of the null dereference risk also indicating that there are other options to handle the null dereference risk.

**Prompting Strategy 2: Prompt Chaining**

Prompt 1: Identify if there are any errors in the following Java code.

Prompt 2: Was the above determined error a null dereference risk.

Prompt 3: If there is a null dereference risk, what is the null dereference risk? And if there was not a null dereference risk, how was this determined?

Output: ‘’’

public int getLength(String s) {

if (s == null) {

return 0; // Handling null safety

}

Return s.length();

‘’’

Analysis: The above output is from the GPT model. This output was selected due to its structured breakdown of the null dereference risk. The result is correct offering a more thorough although slightly repetitive explanation on the null dereference risk.

**Model Comparison and Analysis**

Both outputs are fundamentally correct with virtually no difference across models for this software development task despite differences in wording. The Self-Consistency Prompting strategy appeared to offer more robust alternatives to fixing the null dereference risk; however, these responses were filtered out due to averaging the prompt outputs together. The Prompt Chaining strategy achieved a more descriptive explanation as to the reasoning of the null dereference risk and how to fix it; however, this also came with slight increases in issues concerning redundancy.

1. Task: **CSV Parser Variants (Python).** Improve the parser to support quoted fields.

def parse\_csv\_line(line):

return line.split(‘,’) # Incomplete: doesn’t handle quoted fields

**Prompting Strategy 1: Zero-Shot Prompting**

Prompt: Improve the Python code for the following parser implementation to support quoted fields.

Output: ‘’’

import csv

def parse\_csv\_line(line);

return next(csv.reader([line]))

‘’’

Analysis: The above output is from the GPT model. This output was selected because it is syntactically simple and addresses the quoted field issue efficiently. This result adheres to handling the CSV cases despite its minimal nature making use of the CSV cases.

**Prompting Strategy 2: Chain-of-Thought Prompting**

Prompt: Improve the Python code for the following parser to support quoted fields explaining the reasoning of each step in the implementation.

Output: ‘’’

def parse\_csv\_line(line):

result = []

current\_field = []

inside\_quotes = False

i = 0

while i < len(line):

char = line[i]

if char == “” and (i == 0 or line[i-1] != ‘\\’):

inside\_quotes = not inside\_quotes

i += 1

continue

if char == ‘,’ and not inside\_quotes:

result.append(‘’.join(current\_field))

current\_field = []

else:

current\_field

i += 1

result.append(‘’.join(current\_field))

return result

‘’’

Analysis: The above output is from the Claude model. This output was selected because it provided thorough step-by-step breakdown of character-by-character parsing accounting for quoted fields. The result is correct explaining and implementing the inner workings of a typical CSV parser.

**Model Comparison and Analysis**

Both outputs are fundamentally correct. The Zero-Shot Prompting strategy was more likely to use a built in library to parse the data while the Chain-of-Thought Prompting strategy was able to explain the implementation more effectively using the step-by-step framework. However, the implementation by the character basis was only completed in Claude and GPT for Chain-of-Thought Prompting.

1. Task: **Data Class to API Conversion (Kotlin).** Convert the data class to a REST API for Ktor.

data class Product(val id: Int, val name: String, val price: Double)

// TODO: Create GET and POST endpoints using Ktor

**Prompting Strategy 1: Chain-of-Thought Prompting**

Prompt: Convert the following Kotlin data class to a REST API for Ktor explaining the reasoning of each step in the implementation.

Output: ‘’’

import io.ktor.application.\*

import io.ktor.features.ContentNegotiation

import io.ktor.http.\*

import io.ktor.response.\*

import io.ktor.request.\*

import io.ktor.routing.\*

import io.ktor.serialization.\*

import io.ktor.server.engine.\*

import io.ktor.server.netty.\*

data class User(val id: Int, val name: String)

fun main() {

embeddedServer(Netty, port = 8080) {

install(ContentNegotiation) {

json()

}

routing {

get("/user") {

call.respond(User(1, "Alice"))

}

}

}.start(wait = true)

}

‘’’

Analysis: The above output is from the Claude model. The output was selected because it provides a complete explanation of how to convert a Kotlin data class into a working REST API using Ktor. The result is correct showcasing each stage coherently and is syntactically robust.

**Prompting Strategy 2: Self-Consistency Prompting**

Prompt: Convert the following Kotlin data class to a REST API for Ktor.

The above prompt was passed 5 times.

Output: ‘’’

import io.ktor.application.\*

import io.ktor.response.\*

import io.ktor.request.\*

import io.ktor.routing.\*

import io.ktor.server.engine.\*

import io.ktor.server.netty.\*

import io.ktor.features.\*

import kotlinx.serialization.Serializable

import kotlinx.serialization.json.Json

import io.ktor.serialization.\*

@Serializable

data class User(val id: Int, val name: String)

fun main() {

embeddedServer(Netty, port = 8080) {

install(ContentNegotiation) {

json()

}

routing {

get("/user") {

call.respond(User(1, "Bob"))

}

}

}.start(wait = true)

}

‘’’

Analysis: The above output is from the Gemini model. The output was selected based on its syntactic robustness. The result is correct, but lacks descriptive reasoning in its choices.

**Model Comparison and Analysis**

All models were able to output nearly identical implementations of this software task with correct Kotlin/Ktor. Claude appeared to have the most descriptive output as compared to the other models. The Chain-of-Though Prompting strategy also produced more descriptive results as to be expected with the nature of the step-by-step suggestion in the prompts. The Self-Consistency Prompting strategy code in contrasts lacked this elaboration but was still able to complete the task correctly.

1. Task: **Function Summarization (Python).** Write a brief summary of the function.

def reverse\_words(sentence):

return ‘ ‘.join(sentence.split()[::-1])

**Prompting Strategy 1: Chain-of-Thought Prompting**

Prompt: Write a brief summary of the following Python function explaining the step-by-step reasoning for how you conclude that summary.

Output: I'll analyze this Python function step-by-step. The function is called reverse\_words. It receives a sentence (a string), splits it into a list of words using split(), reverses the list using [::-1], and then joins the reversed list back into a string using " ".join().  
Therefore, the function returns the sentence with the order of the words reversed, while keeping the words themselves intact.

Summary: The function takes a sentence and returns it with the order of words reversed.

Analysis: The above output is from the Claude model. The output was selected based on its concise explanation of the function. The result is a correct summary completed in a singular sentence that still has the explanation associated with the step-by-step portion of the prompt.

**Prompting Strategy 2: Role-Playing Prompting**

Prompt: You are an expert Python programmer that wrote the following Python code.

You have been tasked with providing a summarized report of your implementation.

Output: This function accepts a sentence as input. It first splits the sentence into words using Python’s built-in split() method. It then reverses the list of words using slicing ([::-1]) and finally joins the reversed words back into a string using " ".join().  
The resulting string maintains the original words but in reverse order.  
**Purpose:** To reverse the word order in a sentence while preserving individual word content.

Analysis: This output is from the GPT model. The output was selected for its well-structured explanation and appropriate documentation. The result is a correct output that mimics the technical language effectively.

**Model Comparison and Analysis**

Both outputs accurately summarize the purpose and inner workings of the function in about a sentence. The Chain-of-Though strategy effectively summarized the function in natural language while the Role-Playing Prompting strategy contained more technical based language. The difference between models was negligible. And overall, both strategies worked effectively with the best prompt being a stylistic preference.

1. Task: **Prompt from Code Comments (Python).** Write a prompt that could generate the code.

# This function checks if a number is prime

def is\_prime(n):

if n <= 1:

return False

for i in range(2, int(n\*\*0.5)+1):

if n % i == 0:

return False

return True

**Prompting Strategy 1: Zero-Shot Prompting**

Prompt: Write a prompt that could generate the following Python code.

Output: Write a Python function called is\_prime that takes one integer argument n. The function should return True if n is a prime number and False otherwise. Implement the primality test by checking for divisibility from 2 up to the square root of n. Handle the base cases where n is less than or equal to 1. Include comments to explain the logic.

Analysis: The above output is from the Llama model. The output was selected due to the direct nature of the prompt, demonstrating a complete understanding of the code. The result is correct but did not make use of a specific prompting strategy.

**Prompting Strategy 2: Chain-of-Thought Prompting**

Prompt: Write a prompt that could generate the following Python code explaining the step-by-step reasoning for the prompt engineering approach.

Output: Generate Python code for a function called `is\_prime` that takes one integer argument `n` and returns `True` if `n` is a prime number, and `False` otherwise.

Analysis: The above output is from the Gemini model. The output was selected due to the concise nature of the prompt and being reminiscent of a Zero-Shot Prompting strategy. The result is correct and mirrors a prompt engineer strategy developers use despite being a simplistic one.

**Model Comparison and Analysis**

Both outputs were fundamentally correct. Each model except for Gemini generated descriptive prompts that would perform the task effectively. However, the prompts bordered on convoluted with the amount of detail given. The Gemini model was able to generate a prompt that could be used by developers in the Chain-of-Thought prompting strategy. It also provided a well-written explanation of the methodology behind generating the prompt. The Zero-Shot Prompting strategy was more robust than expected despite the nondescript prompt.

1. Task: **Fixing Factorial Bug (Python).** Fix the bug when input is 0.

def factorial(n):

result = 1

for i in range(1, n):

result \*= i

return result

**Prompting Strategy 1: Self-Consistency Prompting**

Prompt: Fix the bug of the following Python code for when the input is 0.

The above prompt was passed 5 times.

Output: ‘’’

def factorial(n):

result = 1

for i in range(1, n + 1): # Changed to include n itself

result \*= i

return result

‘’’

Analysis: The above output is from the Claude model. The output was selected due to its effective handling of the explicitly error. The result is correct and demonstrated a response to the task instead of jumping ahead like other models.

**Prompting Strategy 2: Chain-of-Thought Prompting**

Prompt: Fix the bug of the following Python code for when the input is 0 explaining the step-by-step reasoning for your implementation.

Output: ‘’’

def factorial(n):

if n == 0: # Step 1: Handle the special case where n is 0

return 1 # By definition, 0! = 1

result = 1

for i in range(1, n + 1): # Step 2: Loop correctly from 1 to n

result \*= i

return result

‘’’

Analysis:

**Model Comparison and Analysis**

Both outputs are fundamentally correct. The models tended to jump ahead debugging the code entirely instead of the explicit task. The Claude model was the only model to only debug “when the input is 0”. The rest of the model’s output also tended to be overly complex as seen in the Chain-of-Thought Prompting strategy. Thus, the Self-Consistency Prompting Strategy appeared to be more effective in this task.

1. Task: **Linked List Node Deletion (C).** Implement node deletion by value.

struct Node {

int data;

struct Node\* next;

};

void deleteNode(struct Node\*\* head, int key) {

// TODO: Implement node deletion

}

**Prompting Strategy 1: Zero-Shot Prompting**

Prompt: Implement node deletion by value in C using the following C framework to do so.

Output: ‘’’

void deleteNode(struct Node\*\* head\_ref, int key) {

struct Node\* temp = \*head\_ref, \*prev;

if (temp != NULL && temp->data == key) {

\*head\_ref = temp->next;

free(temp);

return;

}

while (temp != NULL && temp->data != key) {

prev = temp;

temp = temp->next;

}

if (temp == NULL) return;

prev->next = temp->next;

free(temp);

}

‘’’

Analysis:

**Prompting Strategy 2: Role-Playing Prompting**

Prompt: You are an expert software developer and master of coding in C. You have been tasked with completing the following C framework to implement node deletion by value in C.

Output: ‘’’

void deleteNode(struct Node\*\* head\_ref, int key) {

struct Node\* temp = \*head\_ref, \*prev = NULL;

while (temp != NULL && temp->data != key) {

prev = temp;

temp = temp->next;

if (temp == NULL) return;

if (prev == NULL)

\*head\_ref = temp->next;

else

prev->next = temp->next;

free(temp);

}

‘’’

Analysis: The above output is from the GPT model. The output was selected due to its logical correctness and efficiency. The result avoids any code duplication and handles all cases.

**Model Comparison and Analysis**

Both outputs were correct and nearly identical in logic. The Role-Playing Prompting strategy resulted in slightly more optimized code compared to the Zero-Shot Prompting. However, Claude was marginally more readable across tasks due to separation of cases.

1. Task: **Recursive Function Completion (Python).** Complete the recursive function for Fibonacci.

def fibonacci(n):

# TODO: Base cases and recursive call

pass

**Prompting Strategy 1: Role-Playing Prompting**

Prompt: You are an expert Python programmer who has mastered implementation of recursive functions. You are tasked with completing the recursive following Python function for the Fibonacci sequence.

Output: ‘’’

def fibonacci(n):

if n <= 0:

return 0

elif n == 1:

return 1

else:

return fibonacci(n-1) + fibonacci(n-2)

‘’’

Analysis: The above output is from the Claude model. This output was selected because it concisely and accurately implements the recursive Fibonacci algorithm. The result is correct with implementation typical of recursive algorithms and proper conditional branching.

**Prompting Strategy 2: Few-Shot Prompting**

Prompt: Code: ‘’’

def factorial(n):

# TODO: base cases and recursive call

pass

‘’’

Recursive Implementation: ‘’’

def factorial(n):

if n = 0 or n == 1:

return 1

else:

return n \* factorial(n-1)

‘’’

Code: ‘’’

def reverse\_string(s):

# TODO: base case and recursive call

pass

‘’’

Recursive Implementation: ‘’’

def reverse\_string(s):

if len(s) <= 1:

return s

else:

return reverse\_string(s[1:]) + s[0]

‘’’

Complete the following implementation of the recursive function for Fibonacci sequence in Python.

Output: ‘’’

def fibonacci(n):

if n <= 0:

return 0

elif n == 1:

return 1

else:

return fibonacci(n-1) + fibonacci(n-2)

‘’’

Analysis: The above output is from the GPT model. This output was selected because it accurately implements the Fibonacci sequence recursively. The result is correct and follows the structure of the provided examples.

**Model Comparison and Analysis**

All models produced functional solutions that were negligibly different. Due to the code heavy nature of both prompting strategies, the outputs were focused and did not differ significantly. The Few-Shot Prompting strategy appears more apt for the task due to lack of natural language needed to complete the task.

1. Task: **Constructor Completion (Python).**  Complete the class constructor.

class Person:

def \_\_init\_\_(self):

# TODO: Add name, age, and optional email

pass

**Prompting Strategy 1: Role-Playing Prompting**

Prompt: You are an expert Python software developer tasked with completing the following constructor class.

Output: ‘’’

class Person:

def \_\_init\_\_(self, name, age, email):

self.name = name

self.age = age

self.email = email

‘’’

Analysis: The above output is from the GPT model. This output was selected because it clearly initializes the object attributes using the given parameters. The result does not fully complete the task because it requires all inputs.

**Prompting Strategy 2: Self-Consistency Prompting**

Prompt: Complete the following class constructor in Python code.

The above prompt was passed 5 times.

Output: ‘’’

class Person:

def \_\_init\_\_(self, name, age, email=None):

self.name = name

self.age = age

self.email = email

‘’’

Analysis: The above output is from the Gemini model. The output was selected because it includes that email is an optional input. The result is syntactically correct.

**Model Comparison and Analysis**

Both outputs generated syntactically valid class constructors. The generations were uniform across the models with only the Gemini model producing an output that completed the entire task. Thus, Gemini is the only model that picked implemented a default parameter for email making it an optional input. The Self-Consistency Prompting strategy was able to pick up on this nuance of the task due to the multiple inputs, showing its better for detailed tasks.

1. Task: **Binary Search Completion (Java).** Complete the 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;

// TODO: Compare and adjust bounds

}

Return -1;

}

**Prompting Strategy 1: Role-Playing Prompting**

Prompt: You are an expert Java developer tasked with completing the following binary search implementation.

Output: ‘’’

public class BinarySearch {

public static int binarySearch(int[] arr, int target) {

int left = 0;

int right = arr.length – 1;

while (left <= right) {

int mid = left + (right – left) / 2;

if (arr[mid] == target) {

return mid;

} else if (arr[mid] < target) {

Left = mid + 1;

} else {

right = mid – 1;

}

return -1;

}

}

‘’’

Analysis: The above output is from the GPT model. This output was selected because it correctly implements an iterative binary search in Java. The result is syntactically sound.

**Prompting Strategy 2: Prompt Chaining**

Prompt 1: Determine what section of the following binary search implementation in Java is not completed.

Prompt 2: Complete the implementation of the section that was found to not be completed above.

Prompt 3: Is the completed implementation of the binary search algorithm correct, if not please correct the Java code.

Output: ‘’’

public class BinarySearch {

public static int binarySearch(int[] arr, int target) {

int left = 0;

int right = arr.length – 1;

while (left <= right) {

int mid = left + (right – left) / 2;

if (arr[mid] == target) {

return mid;

}

if (arr[mid] < target) {

left = mid + 1;

} else {

Right = mid – 1;

}

}

return -1;

}

}

‘’’

Analysis: The above output is from the Claude model. The output was selected because it correctly implements an iterative binary search in Java. The result is syntactically sound.

**Model Comparison and Analysis**

Both outputs are syntactically and logically identical, indicating consistency across models and prompting strategies for this task. The difference in reasoning style in the selected prompting strategies did not appear to have an impact on effectiveness of the implementation.

1. Task: **Self-Consistency Bug Fixing (C++).** Resolve inconsistency between function name and logic.

// Supposed to return true if x is even

bool isOdd(int x) {

return x % 2 == 0; // Logic contradicts function name

}

**Prompting Strategy 1: Few-Shot Prompting**

Prompt:Code: ‘’’

// Supposed to return true if x is odd

bool isEven(int x){

return x % 2 == 1;

}

‘’’

Resolved Code: ‘’’

bool isOdd(int x){

return x % 2 == 1;

}

‘’’

Code: ‘’’

int add(int a, int b) {

return a – b;

}

‘’’

Resolved Code:’’’

int subtract(int a, int b) {

return a – b

}

‘’’

Resolve the inconsistency between function name and logic in the following C++ code using the above examples as a model for how to do so.

Output: \*\*Option 1\*\*

‘’’

bool isEven(int x) {

return x % 2 == 0;

}

‘’’

\*\*Option 2\*\*

‘’’

bool isOdd(int x) {

return x % 2 == 1;

}

‘’’

Analysis: The above output is from the Claude model. This output was selected because it offered two options for resolving the logic inconsistency. The result is syntactically correct and logically matches for both cases.

**Prompting Strategy 2: Self-Consistency Prompting**

Prompt: Resolve the inconsistency between function name and logic in the following C++ code.

The above prompt was passed 5 times.

Output: ‘’’

bool isEven(int x) {

return x $ 2 == 0;

}

‘’’

Analysis: The above output is from the GPT model. This output was selected because it preserves the original functionality and indication of intent from the input code’s comment. The result is correct and logically matches.

**Model Comparison and Analysis**

All models correctly resolve the inconsistency between function name and the logic. The Claude and Gemini models present two options for fixing this inconsistency in the Few-Shot Prompting strategy while the rest of the models only present one in each output. The prompting strategies appeared to be more likely to adjust function name across both strategies. Both strategies had outputs that were fundamentally similar, indicating the strategy in negligible for such a simple task.

1. Task: **Prompt Chaining: Bug Identification** 🡪 **Fix (JavaScript).** Identify and fix the bug.  **Prompting Strategy 1: Prompt Chaining**

function isEven(n) {

return n % 2; // Returns 1 or 0, not true/false

}

Prompt 1: Identify the bug in the following JavaScript code.

Prompt 2: Fix the bug found in the above JavaScript code.

Output: ‘’’

function isEven(n) {

return n % 2 === 0;

}

‘’’

Analysis: The above output is from the GPT model. The output was selected because it fixed the logical error. The result corrects the function so that it returns Booleans.

**Prompting Strategy 2: Role-Playing Prompting**

Prompt: You are an expert software developer well-versed in JavaScript. You have been given the following code and tasked with identifying and fixing the bug found in the code.

Output: ‘’’

function isEven(n) {

return n % 2 === 0;

}

‘’’

Analysis: The above output is from the Claude model. The output was selected because it is concise and correctly fixes the bug. The result is correct yet fundamentally similar to the other output.

**Model Comparison and Analysis**

Both outputs are logically correct and resolve the identified bug by returning Boolean values. The Role-Playing Prompting strategy was able to frame the solution due to its direction to complete it like a software developer, while Prompt Chaining provided a direct correction without narrative. Model performance was nearly identical, likely due to the simplistic nature of the task.

1. Task: **Summary Decomposition (C++).** Decompose the high-level comment/summary into logical steps.

**Prompting Strategy 1: Few-Shot Prompting**

// Function that validates as input, calculates square, and returns result

int process(int x) {

if (x < 0) return -1;

return x \* x;

}

Prompt: Code: ‘’’

// Function that initializes an input as a maximum, checks if the next input is larger than the current maximum, and checks if the next input is larger than the current maximum, and returns the overall determined maximum

int max\_of\_three(int a, int b, int c) {

int max = a;

if (b > max) {

max = b;

}

if (c > max) {

max = c;

}

return max;

‘’’

Decomposed comment: ‘’’

// function to determine maximum of 3 values

int max\_of\_three(int a, int b, int c) {

// initialize maximum

int max = a;

// check if larger than current maximum

if (b > max) {

max = b;

}

// check if larger than current maximum

if (c > max) {

max = c;

}

// returning the maximum

return max;

}

‘’’

Code: ‘’’

// function that initializes container sum, iterates through all values in the array, adding each value to the sum, and returns the computed sum

int sum\_array(int arr[], int size) {

int sum = 0;

for (int i = 0; i < size; i++) {

sum += arr[i];

}

return sum;

‘’’

Decomposed comment:’’’

// function that sums all elements in an array

int sum\_array(int arr[], int size) {

int sum = 0;

for (int i = 0; i < size; ++i) {

sum += arr[i];

}

return sum;

‘’’

Decompose the high-level comment/summary from the following C++ code into logical steps using the above examples as models for how to do so.

Output: ‘’’

// Function that validates input, calculates square, and returns result

int process(int x) {

// Validate input: return -1 if x is negative

if (x < 0) return -1;

// Calculate square of x

return x \* x;

}

‘’’

Analysis: The above output is from the Claude model. This output was selected because it clearly breaks down the high-level summary into actionable and readable steps. The result is correct and has a comment for each line.

**Prompting Strategy 2: Prompt Chaining**

Prompt 1: Identify the high-level comment/summary in the following C++ code.

Prompt 2: Decompose the high-level comment/summary identified above into logical steps.

Output: ‘’’

// Function that validates input, computes square if valid, and returns the result

int process(int x) {

// Step 1: Check if input is negative

if (x < 0) return -1;

// Step 2: Compute square

return x \* x;

}

‘’’

Analysis: The above output is from the GPT model. The output was selected because it translates the initial comment into a logical sequence. The result is correct; however, it does not alter the initial comment.

**Model Comparison and Analysis**

Both outputs are fundamentally correct. The Few-Shot Prompting strategy had more concise and natural comments embedded in the code. The Prompt Chaining emphasized the reasoning of the code and did not alter the initial comment. GPT’s approach resulted in more numbered steps. Claude’s approach was simpler and cleaner. Overall, both strategies completed the task; however, the Few-Shot Prompting strategy appeared to be core effective and concise.

1. Task: **Purpose Inference 🡪 Completion (Python).** Complete the function based on intent.

**Prompting Strategy 1: Zero-Shot Prompting**

def calculate\_average(scores):

total = 0

# TODO: Complete to return average

pass

Prompt: Complete the following Python function based on the commented intent.

Output: ‘’’

def calculate\_average(scores):

total = sum(scores)

return total / len(scores) if scores else 0

‘’’

Analysis: The above output is from the Claude model. This output was selected because it handles the intent of averaging and a safety check in a single line. The result is a correct function that effectively infers the purpose from the function name, comment, and partial implementation.

**Prompting Strategy 2: Few-Shot Prompting**

Prompt: Code: ‘’’

def calculate\_sum(lst):

sum = 0

# TODO: Complete to return sum

pass

‘’’

Completed Code: ‘’’

def calculate\_sum(lst):

sum = 0

for element in lst:

sum = sum + element

return sum

‘’’

Code: ‘’’

def find\_max(numbers):

max\_num = numbers[0]

# TODO: Complete to find maximum numbers

pass

‘’’

Completed Code: ‘’’

def find\_max(numbers):

max\_num = numbers[0]

for num in numbers:

if num > max\_num:

max\_num = num

return max\_num

‘’’

Complete the following Python function based on the commented intent using the above examples to model how to do so.

Output: ‘’’

def calculate\_average(scores):

total = 0

for score in scores:

total += score

return total / len(scores) if len(scores) > 0 else 0

‘’’

Analysis: The above output is from the GPT model. This output was selected because it follows the structure of the examples effectively. The result is a correct implementation that is not as efficient as possible; however, this is likely due to the examples not being entirely robust.

**Model Comparison and Analysis**

Both outputs were fundamentally correct. The difference between the outputs is negligible between models and comes down to prompting strategies. The Zero-Shot Prompting strategy led to a more concise output taking use of the sum() function. The Few-Shot Prompting strategy was not as concise. However, this is likely due to the sum() function not being used in any of the examples and not necessarily indicating the Zero-Shot Prompting strategy is better on this specific task.

1. Task: **Full-File Bug Detection and Refactoring (Python).** Analyze the following utility script. Identify any potential logic issues or design flaws. Then, refactor it for better readability, correctness, and safety.

**Prompting Strategy 1: Prompt Chaining**

# 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()

Prompt 1: Analyze the following utility script implemented in Python.

Prompt 2: Identify any potential logic issues in the above code that was analyzed.

Prompt 3: Identify any potential design flaws in the above code that was analyzed.

Prompt 4: Refactor the above code for better readability based on the above analysis and above identified potential logic issues and design flaws.

Prompt 5: Refactor the above code for better correctness based on the above analysis and above identified potential logic issues and design flaws.

Prompt 6: Refactor the above code for better safety based on the above analysis and above identified potential logic issues and design flaws.

Output:’’’

import csv

def read\_csv(filepath):

try:

with open(filepath, ‘r’) as f:

return [row for row in csv.reader(f)]

except FileNotFoundError:

print(f”Error: File ‘{filepath}’ not found.”)

return []

def summarize\_column(data, index):

try:

values = [float(row[index]) for row in data[1:] if row[index]]

total = sum(values)

avg = total / len(values) if values else 0

return total, avg

except (IndexError, ValueError) as e:

print(f”Error processing column {index}: {e}”)

return 0, 0

def main():

filepath = ‘data.csv’

data = read\_csv(filepath)

if not data:

return

total, avg = summarize\_column(data, 1)

print(f”Total: {total}”)

print(f”Average: {avg}”)

if \_\_name\_\_ == ‘\_\_main\_\_’:

main()

‘’’

Analysis: The above output is from the Claude model. This output was selected because it improves robustness through structured error handling and data parsing issues. The result is a file that handles missing files, empty rows, non-numerical values, while preserving readability.

**Prompting Strategy 2: Chain-of-Thought Prompting**

Prompt: Analyze the following utility script implemented in Python. Identify any potential logic issues or design flaws. Then, refactor it for better readability, correctness, and safety. Explain the step-by-step process and reasoning for each step and implementation.

Output: ‘’’

import csv

def read\_csv(filepath):

“””Reads a CSV file and returns its contents as a list of rows.”””

try:

with open(filepath, ‘r’) as file:

return list(csv.reader(file))

except FileNotFoundError:

print(f”Error: The file ‘{filepath}’ does not exist.”)

return []

def summarize\_column(data, index):

“””Calculates the total and average of a numeric column in a dataset.”””

try:

values = [float(row[index]) for row in data[1:] if row[index].strip()]

if not values:

return 0, 0

total total, total / len(values)

except (IndexError, ValueError) as e:

print(f”Data error in column {index}: {e}”)

return 0, 0

def main():

filepath = ‘data.csv’

data = read\_csv(filepath)

if data:

total, average = summarize\_column(data, 1)

print(f”Total: {total}”)

print(f”Average: {average}”)

if \_\_name\_\_ == “\_\_main\_\_”:

main()

‘’’

Analysis: The above output is from the GPT model. This output was selected to improve safety with error handling for missing or malformed input. The result is well structured containing adequate comment notation and adequate rationale.

**Model Comparison and Analysis**

Both outputs effectively improve robustness and clarity. The Chain-of-Thought Prompting strategy provided explanatory information focusing on documentation and maintainability. The Prompt Chaining is more effectively at modularizing improvements but slightly more fragmented in its approach due to the sequence of prompts.

1. Task: **Code Completion and Robustness Enhancement (Python).** Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly.

**Prompting Strategy 1: Self-Consistency Prompting**

# 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()

Prompt: Complete the following file-processing script implemented in Python, reaching the goals of cleaning each line, removing punctuation, and counting word frequencies correctly.

The above prompt was passed 5 times.

Output: ‘’’

import string

def load\_file(filepath):

with open(filepath, ‘r’) as f:

return f.readlines()

def clean\_line(line):

return line.translate(str.maketrans(‘’, ‘’, string.punctuation)).lower()

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()

‘’’

Analysis: The above output is from the Claude model. This output was selected because it completes the script using built—in libraries. The result correctly handles file reading, case normalization and tokenization in a clean and readable structure.

**Prompting Strategy 2: Prompt Chaining**

Prompt 1: Analyze the incomplete implementation of the following file-processor implemented in Python.

Prompt 2: Implement the section of the above code for removing the punctuation.

Prompt 3: Implement the section of the above code for cleaning each line.

Prompt 4: Implement the section of the above code for counting word frequencies correctly.

Prompt 5: Analyze and correct the code for consistency between the original and the above implementation tasks.

Output: ‘’’

import string

def load\_file(filepath):

with open(filepath, ‘r’) as file:

return file.readlines()

def clean\_line(line):

cleaned = line.strip().translate(str.maketrans(‘’, ‘’, string.punctuation))

return cleaned.lower()

def count\_words(lines):

word\_counts = {}

for line in lines:

clean = clean\_line(line)

words = clean.split()

for word in words:

word\_counts[word] = word\_counts.get(word, 0) + 1

return word\_counts

def main():

filepath = ‘input.txt’

lines = load\_file(file\_path)

if not lines:

print(“No data to process.”)

return

counts = count\_words(lines)

for word, count in sorted(counts.items()):

print(f”{word}: {count}”)

if \_\_name\_\_ == ‘\_\_main\_\_’:

main()

‘’’

Analysis: The above output is from the GPT model. The output was selected because it demonstrates both correctness and attention to practical safety measures. The result correctly finishes the implementation, adding input sanitation, punctuation removal and lowercase normalization.

**Model Comparison and Analysis**

Both outputs produced nearly identical and correct solutions. The Prompt Chaining strategy produced slightly more input validation and defensive programming. The Self-Consistency Prompting strategy was more focused on linear structure and slightly more efficient for this type of task.