GenAI for Software Development: Assignment 3

Benjamin Tremblay
bptremblay@wm.edu

Rowan Miller
rvmiller@wm.edu

1 Introduction

Throughout the semester, we have built up to prompt engineering, and the different strategies that can be used to prompt Large Language Models. In this assignment, we compare multiple AI models using several prompt engineering strategies to evaluate the effectiveness of said strategies.

2 Dataset Preparation

We chose 5 different AI models to use in our experiment: gpt-4o-mini, Codestral-2501, Llama-4-Scout-17B-16E-Instruct, Ministral-3B, and gemini-2.0-flash. This allows for a wider look at the AI landscape, and chance to analyze modern models.

If not otherwise stated, the temperature used in prompting is .7. This provides a standard basis to test lower and higher temperatures, and lets us test the AIs on an even playing field. The .ipynb files in the GitHub repository were used to produce the prompt responses and gather our information. We constructed a CSV file called Problems_Explored where we gathered our data in an organized way, which we will refer to during the analysis.

3 Analysis

Using the Large Language Models mentioned above, we tried a variety of prompt engineering strategies to answer various questions that one might ask to get an idea of the effectiveness of each prompt. Considering that there is no standardized answer to compare the LLMs' answers with, we will provide an analysis for each problem, comparing prompting strategies and models to each other.

Name	Problem 1		
Goal	Summarize a given Java method		
Model	gpt-4o-mini	**Method**	Zero Shot
Temperature	0.7	**Token Limit**	1024
Prompt	Summarize the functionality of the following Java method. Be concise and to the point: 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 ;
}
```

Output	The `countWordFrequency` method takes a list of strings (`words`) and returns a map that counts the frequency of each unique word. It uses a `HashMap` to store words as keys and their corresponding counts as values, updating the count for each word as it iterates through the list.

Analysis

We used GPT and Gemini, and tried a zero shot prompt and a roleplay one where the model was cast as an engineer. The roleplay had little to no effect on either model's output, and all 4 prompts produced good results. Ultimately, GPT and zero shot were chosen as the optimal combination due to their clarity. The other results can be found in rows 2 and 3 of Problems_Explored.csv.

Name	Problem 2		
Goal	Fix an off-by-one error in a Python function		
Model	gpt-4o-mini	**Method**	Roleplay
Temperature	0.7	**Token Limit**	1024
Prompt	You are an experienced Software Engineer adept at bugfixing. Identify and fix the off-by-one error in the following python function. Be concise and to the point:def sum_range (start , end) :		

```
  total = 0
  for i in range ( start , end ) :
    total += i
  return total
```

Output	The off-by-one error in the function arises because the `end` parameter is exclusive in the `range` function, meaning the sum does not include the `end` value. To include `end` in the sum, you should adjust the range to `range(start, end + 1)`. Here is the corrected function: \`\`\`python

~~~
def sum_range(start, end):
    total = 0
    for i in range(start, end + 1):
        total += i
    return tot
```
~~~

Analysis

We used GPT and Codestral, again using zero shot and roleplay. Similarly to Problem 1, all 4 outputs produced good results. GPT and roleplay were chosen as the optimal combination because of the clarity of its explanation, explaining how the range function works. The other results can be found on rows 4 and 5 of Problems_Explored.csv.

Name	Problem 3		
Goal	Classify a C++ bug		
Model	gpt-4o-mini	**Method**	Few Shot
Temperature	0.7	**Token Limit**	1024

Prompt

Classify the type of bug which appears in the following C++ function. Be concise. Here's an example of what the code should do: given 2: return a pointer to an integer array of size 2. Given 6: return a pointer to an integer array of size 6.

```
int * getArray ( int size ) {
  int arr [ size ]; // Warning : local array
  return arr ; // Bug : returning pointer to local variable
}
```

Output

The bug is a **dangling pointer** issue. The function returns a pointer to a local array that goes out of scope when the function exits, leading to undefined behavior.

Analysis

We used GPT and Gemini, using zero shot and few shot. The difference between zero shot and few shot was negligible; Gemini's output to the two prompts was effectively identical. We chose GPT and few shot as the optimal combination, as it concisely explained why a dangling pointer causes problems. The other results can be found on rows 6 and 7 of Problems_Explored.csv.

Name	Problem 4		
Goal	Validate email addresses with Regex		
Model	Codestral-2501	**Method**	Zero Shot
Temperature	0.7	**Token Limit**	1024
Prompt	Complete the following python function by using regex to validate basic email addresses: def is_valid_email (email) : # TODO : Complete using regex pass		

Output

To validate basic email addresses using a regular expression in Python, you can use the `re` module. Below is a completed version of the `is_valid_email` function that uses a regex pattern to validate the email address:

```python
import re

def is_valid_email(email):
    # Define the regex pattern for validating an email address
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    # Use the re.match function to check if the email matches the pattern
    if re.match(pattern, email):
        return True
    else:
        return False

# Example usage
print(is_valid_email("test@example.com"))  # True
print(is_valid_email("invalid-email@"))    # False
```

### Explanation of the Regex Pattern:
- `^[a-zA-Z0-9._%+-]+`: Ensures the email starts with one or more alphanumeric characters, dots, underscores, percent signs, plus signs, and hyphens.
- `@[a-zA-Z0-9.-]+`: Ensures the email contains an `@` symbol followed by one or more alphanumeric characters, dots, and hyphens.
- `\.[a-zA-Z]{2,}$`: Ensures the email ends with a dot followed by two or more alphabetic characters (this is a basic check for the domain suffix).

	This pattern covers many common email formats but may not be exhaustive. Depending on your needs, you might want to refine the pattern or add additional checks.

Analysis

We used GPT and Codestral, again using zero shot and few shot prompts. All 4 results were again satisfactory, but the zero shot outputs provided a more complete explanation of the code. Both Codestral and GPT had strikingly similar responses, even producing a nearly identical clarification at the end of their output that their regex solutions were inexhaustive. We determined the Codestral zero shot combination to be optimal due to its clarity and robust explanation. The other prompts can be found on lines 8 and 9 of Problems_Explored.csv.

Name	Problem 5		
Goal	Create a Flask endpoint that returns a JSON greeting		
Model	gemini-2.0-flash	**Method**	Chain of Thought
Temperature	0.7	**Token Limit**	1024

Prompt

Create a '/greet/<username>' endpoint that returns a JSON greeting. Break down your reasoning step-by-step.

```
from flask import Flask , jsonify

app = Flask ( __name__ )

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

Output

Okay, let's break down how to create the `/greet/<username>` endpoint using Flask and return a JSON greeting.

1. Understanding the Requirements

*   **Endpoint:** `/greet/<username>` This means the URL will look like `/greet/Alice`, `/greet/Bob`, etc. The `<username>` part is a variable that Flask will capture.
*   **JSON Response:** The endpoint must return data in JSON format. This is a standard way for web services to send data to clients.
*   **Greeting:** The JSON should contain a greeting that includes the provided username.

2. Implementing the Flask Route

We already have the basic Flask app setup and the route defined:

```python
from flask import Flask, jsonify

app = Flask(__name__)

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

3. Creating the JSON Response

Inside the `greet` function, we need to:

a. **Construct the greeting message:** We'll use the `username` variable to create a personalized greeting.
b. **Create a Python dictionary:** JSON is based on key-value pairs, so we'll create a Python dictionary to represent our JSON structure.
c. **Use `jsonify`:** The `jsonify` function from Flask converts a Python dictionary into a JSON response with the correct MIME type (`application/json`).

4. Code Implementation

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/greet/<username>')
def greet(username):
    greeting_message = f"Hello, {username}!"  # Create the greeting
    response_data = {'message': greeting_message}  # Create a dictionary
    return jsonify(response_data)  # Convert to JSON and return
```

5. Explanation

* `greeting_message = f"Hello, {username}!"`: This line uses an f-string (formatted string literal) to create the greeting message. The `username`

variable is inserted directly into the string.
*   `response_data = {'message': greeting_message}`: This creates a Python dictionary with a single key, `'message'`, and the value is our greeting message. This is the data that will be converted to JSON.
*   `return jsonify(response_data)`: This line does the magic. `jsonify` takes the `response_data` dictionary and turns it into a Flask `Response` object that contains the JSON representation of the dictionary and sets the `Content-Type` header to `application/json`. This tells the client (e.g., a web browser or another application) that the response is in JSON format.

Complete Code:

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/greet/<username>')
def greet(username):
    greeting_message = f"Hello, {username}!"
    response_data = {'message': greeting_message}
    return jsonify(response_data)

if __name__ == '__main__':
    app.run(debug=True)  # Run the app in debug mode for development
```

How to Run and Test:

1. **Save:** Save the code as a Python file (e.g., `app.py`).
2. **Run:** Open a terminal, navigate to the directory where you saved the file, and run the script using `python app.py`.
3. **Test:** Open a web browser or use a tool like `curl` or `Postman` and go to the URL `http://127.0.0.1:5000/greet/YourName` (replace `YourName` with an actual name). You should see the JSON response:

```json
{
  "message": "Hello, YourName!"
}
```

Important Notes:

*   **`debug=True`:** The `app.run(debug=True)` line is useful during development. It automatically reloads the server when you make changes to

	the code and provides more detailed error messages. Don't use `debug=True` in a production environment. * **Error Handling:** For a real-world application, you would want to add error handling (e.g., what happens if the username is invalid?). * **Deployment:** When you deploy your Flask app to a production server, you'll need to use a WSGI server like Gunicorn or uWSGI.

Analysis

We used GPT and Gemini, using Chain of Thought and Roleplay prompts. The use of Chain of Thought made outputs dramatically longer, and in the case of GPT, somewhat repetitive. We determined Gemini and CoT to be the optimal combination, as it provided an extremely robust explanation of how its code functions without being redundant. The other prompts can be found on lines 10 and 11 of Problems_Explored.csv.

Name	Problem 6		
Goal	Design an SQL schema		
Model	gpt-4o-mini	**Method**	Chain of Thought
Temperature	0.7	**Token Limit**	1024
Prompt	Given the following list of specifications, write the schema for a review app with users, books and reviews. Do this in a step by step process. Give a brief explanation of the code: -- TODO : Design schema with appropriate keys and constraints -- Tables : users (id , name) , books (id , title) , reviews (id , user_id , book_id , rating)		

Output

Let's create a relational database schema for a review app that includes users, books, and reviews. We'll define three tables: `users`, `books`, and `reviews`. I'll walk you through the steps and explain the code along the way.

### Step 1: Define the Users Table

The `users` table will store information about the users of the app. We'll define the columns and set up primary keys and constraints.

```sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,  -- Auto-incrementing ID for each user
    name VARCHAR(255) NOT NULL  -- Name of the user, cannot be null
);
```

Explanation:
- `id SERIAL PRIMARY KEY`: This creates an auto-incrementing primary key for the `users` table, ensuring each user has a unique identifier.
- `name VARCHAR(255) NOT NULL`: This defines a column for the user's name, with a maximum length of 255 characters. The `NOT NULL` constraint ensures that a user must have a name.

### Step 2: Define the Books Table

The `books` table will store information about the books. We'll similarly define the necessary columns and keys.

```sql
CREATE TABLE books (
    id SERIAL PRIMARY KEY,  -- Auto-incrementing ID for each book
    title VARCHAR(255) NOT NULL  -- Title of the book, cannot be null
);
```

Explanation:
- `id SERIAL PRIMARY KEY`: Creates a unique identifier for each book in the same way as for users.
- `title VARCHAR(255) NOT NULL`: Defines a column for the book title, which also cannot be null.

### Step 3: Define the Reviews Table

The `reviews` table will associate users with books and include a rating for each review. It will reference the `users` and `books` tables, enforcing foreign key constraints.

```sql
CREATE TABLE reviews (
    id SERIAL PRIMARY KEY,  -- Auto-incrementing ID for each review
    user_id INT NOT NULL,  -- Foreign key referencing users
    book_id INT NOT NULL,  -- Foreign key referencing books
    rating INT CHECK (rating >= 1 AND rating <= 5),  -- Rating must be between 1 and 5
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,  -- Ensures user exists
    FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE  -- Ensures book exists
);
```

Explanation:
- `id SERIAL PRIMARY KEY`: Again, creates a unique identifier for each review.
- `user_id INT NOT NULL`: This column will store the user ID and cannot be null.
- `book_id INT NOT NULL`: This column will store the book ID and cannot be null.
- `rating INT CHECK (rating >= 1 AND rating <= 5)`: This constraint ensures that the rating must be an integer between 1 and 5.
- `FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE`: This creates a foreign key relationship with the `users` table. If a user is deleted, all their reviews will also be deleted.
- `FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE`: Similar to the user foreign key, this ensures that if a book is deleted, all associated reviews will also be deleted.

### Complete Schema

Putting it all together, the complete schema looks like this:

```sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE books (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL
);

CREATE TABLE reviews (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    book_id INT NOT NULL,
    rating INT CHECK (rating >= 1 AND rating <= 5),
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE
CASCADE,
    FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE
CASCADE
);
```

### Summary

	This schema defines three tables to manage users, books, and reviews, including appropriate keys and constraints to maintain data integrity. The use of foreign keys ensures that relationships between users, books, and their reviews are properly enforced. The checks on the rating ensure valid input, enhancing the reliability of the review system.

Analysis
We used GPT and Codestral, using zero shot and CoT prompts. When the zero shot prompt was given, both GPT and Codestral provided the complete code followed by an explanation. When CoT was used, GPT slotted its explanation in between chunks of code. Codestral, in contrast, provided redundant explanations of its code both before and after it. We determined GPT and CoT to be the optimal combination, due to it successfully breaking a difficult topic into manageable chunks. The other outputs can be found on lines 12 and 13 of Problems_Explored.csv.

Name	Problem 7		
Goal	Correctly identify null dereference risk		
Model	gemini-2.0-flash	**Method**	Zero Shot
Temperature	0.7	**Token Limit**	1024

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

Output	You are absolutely correct! There is a null dereference risk in the provided code. **Explanation:** * **\`s.length()\`: The Problem** * If the input \`String s\` is \`null\`, attempting to call the \`length()\` method on it will result in a \`NullPointerException\`. You are trying to access a member (\`length()\`) of a non-existent object (\`null\`). **How to Fix It:** You need to add a null check *before* attempting to use the \`String s\`. Here are a few common ways to do it:		

1. **Check for Null and Return a Default Value (e.g., 0):**

   ```java
   public int getLength(String s) {
       if (s == null) {
           return 0; // Or some other appropriate default length.
       }
       return s.length();
   }
   ```

   This is a good approach if returning 0 (or another sentinel value) makes sense in the context of your program when the string is null.

2. **Check for Null and Throw an Exception:**

   ```java
   public int getLength(String s) {
       if (s == null) {
           throw new IllegalArgumentException("Input string cannot be null."); // Or a more specific exception
       }
       return s.length();
   }
   ```

   This is suitable if `null` as input represents an invalid state that should halt execution and signal an error. Choose an exception type that best reflects the nature of the problem (e.g., `NullPointerException`, `IllegalArgumentException`).

3. **Using Java 8+ `Optional` (More Advanced):**

   ```java
   import java.util.Optional;

   public int getLength(String s) {
       return Optional.ofNullable(s)
                      .map(String::length)
                      .orElse(0); // Or throw an exception here if that's more appropriate
   }
   ```

   This is a more functional style. `Optional.ofNullable(s)` creates an

	`Optional` object that wraps the string `s`. If `s` is null, the `Optional` will be empty. `map(String::length)` applies the `length()` method to the string *only if* the `Optional` contains a non-null value. `orElse(0)` provides a default value (0 in this case) if the `Optional` is empty (i.e., if `s` was null). **Which Approach to Choose?** The best approach depends entirely on the context of your application and what you want to happen when a null string is passed to the function. * If a null string is a valid possibility and you can handle it gracefully by returning a default length, then the first option (returning 0) is usually the simplest and most appropriate. * If a null string represents a critical error, then throwing an exception is the right choice. * `Optional` is good for more complex scenarios where you want to chain operations and handle potential nulls in a more fluent way. **In summary:** Always be mindful of potential null values when working with object references in Java (and many other languages). Adding null checks is a fundamental part of writing robust and reliable code.

Analysis

We used GPT and Gemini, again using zero shot and CoT prompts. Of these combinations, only Gemini with zero shot properly accounted for the numerous valid options that exist for fixing this bug. However, all 4 options produced satisfactory results. The other outputs can be found on lines 14 and 15 of Problems_Explored.csv.

Name	Problem 8		
Goal	Improve the flexibility of a Python csv parser		
Model	Codestral	**Method**	Chain of Thought
Temperature	0.2	**Token Limit**	1024

Prompt	Improve the following parser method in Python to support quoted fields. Do this step by step:
	def parse_csv_line (line) :
	return line . split (‘,’)		

Output	To improve the `parse_csv_line` method to support quoted fields, we need to handle cases where fields are enclosed in quotes and may contain commas		

within them. Here's a step-by-step approach to achieve this:

### Step 1: Initialize Variables
We'll need to initialize variables to keep track of the current field, whether we are inside quotes, and the result list of fields.

### Step 2: Iterate Through the Line
We'll iterate through each character in the line and build the fields accordingly.

### Step 3: Handle Quoted Fields
We need to handle cases where fields are enclosed in quotes and may contain commas within them. We'll use a flag to keep track of whether we are inside quotes.

### Step 4: Handle Escaped Quotes
We need to handle cases where quotes are escaped within a quoted field.

### Step 5: Handle Multiple Quotes
We need to handle cases where there are multiple quoted fields in a single line.

Here's the improved `parse_csv_line` method:

```python
def parse_csv_line(line):
    fields = []
    current_field = []
    inside_quotes = False
    escape_next = False

    for char in line:
        if escape_next:
            current_field.append(char)
            escape_next = False
        elif char == '"':
            if inside_quotes:
                if len(current_field) > 0 and current_field[-1] == '"':
                    current_field.append(char)
                else:
                    inside_quotes = False
            else:
                inside_quotes = True
                current_field.append(char)
        elif char == ',':
            if inside_quotes:
```

~~~
                current_field.append(char)
            else:
                fields.append(".join(current_field).strip(""))
                current_field = []
        elif char == '\\':
            escape_next = True
        else:
            current_field.append(char)

    # Add the last field
    fields.append(".join(current_field).strip(""))

    return fields

# Example usage:
line = 'name,"John Doe",age,25,"New York, NY"'
print(parse_csv_line(line))
```
~~~

### Explanation:
1. **Initialization**: We initialize `fields` to store the parsed fields, `current_field` to build the current field, `inside_quotes` to track if we are inside quotes, and `escape_next` to handle escaped characters.
2. **Iteration**: We iterate through each character in the line.
3. **Handling Quotes**: If we encounter a quote (`"`), we toggle the `inside_quotes` flag. If we are inside quotes and encounter another quote, we handle it appropriately.
4. **Handling Commas**: If we encounter a comma (`,`) and we are not inside quotes, we finalize the current field and start a new one.
5. **Handling Escapes**: If we encounter a backslash (`\`), we set `escape_next` to handle the next character as an escaped character.
6. **Finalizing Fields**: After the loop, we add the last field to the `fields` list.

This method should correctly handle quoted fields, including those with commas and escaped quotes.

Analysis

We used GPT and Codestral with Chain of Thought prompting. We experimented with a higher temperature of 0.9 and a lower temperature of 0.2. With both temperatures, GPT simply used a library to parse the csv. This is undeniably effective, but misses the point of the prompt. Codestral's response was similar with both high and low temperatures, but was more concise on the low temperature. We chose low-temperature Codestral as the optimal combination. The other outputs can be found on lines 16 and 17 of Problems_Explored.csv.

Name	Problem 9		
Goal	Convert a given data class to REST API using Ktor		
Model	gpt-4o-mini	**Method**	Zero Shot
Temperature	0.7	**Token Limit**	1024
Prompt	Convert the following 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		

Output

~~~
To convert the given `Product` class into a REST API using Ktor, you'll need to set up Ktor in your Kotlin project, create routes for handling GET and POST requests, and manage a simple in-memory storage for the products. Here's a step-by-step implementation:

### Step 1: Set Up Ktor

Make sure you have the Ktor dependencies included in your `build.gradle.kts` file:

```kotlin
dependencies {
    implementation("io.ktor:ktor-server-core:2.x.x")
    implementation("io.ktor:ktor-server-netty:2.x.x")
    implementation("io.ktor:ktor-gson:2.x.x") // For JSON serialization
    testImplementation("io.ktor:ktor-server-tests:2.x.x")
}
```

Replace `2.x.x` with the latest version of Ktor.

### Step 2: Create the Ktor Application

Here's a complete example of how to create a simple Ktor server with GET and POST endpoints for the `Product` class:

```kotlin
import io.ktor.application.*
import io.ktor.features.ContentNegotiation
import io.ktor.gson.gson
import io.ktor.http.HttpStatusCode
~~~

```
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.server.engine.embeddedServer
import io.ktor.server.netty.Netty
import io.ktor.request.receive

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

val products = mutableListOf<Product>()

fun main() {
    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            gson { }
        }
        routing {
            get("/products") {
                // Return the list of products
                call.respond(products)
            }

            post("/products") {
                // Receive a product from the request body
                val product = call.receive<Product>()
                products.add(product)
                call.respond(HttpStatusCode.Created, product)
            }
        }
    }.start(wait = true)
}
```

### Explanation of the Code:

1. **Dependencies**: The example uses Gson for JSON serialization, which allows you to handle JSON requests and responses easily.

2. **Product Data Class**: This is the same product class you provided.

3. **In-Memory Storage**: We're using a mutable list `products` to store `Product` instances.

4. **Ktor Server**:
   - We create an embedded server using Netty.
   - We install the `ContentNegotiation` feature to handle JSON serialization.
   - We define routes:

	- **GET /products**: Responds with the list of all products.
	- **POST /products**: Receives a `Product` object from the request body, adds it to the list, and responds with the created product.
	
	### Step 3: Running the Server
	
	Run the application, and you will have a REST API running on `http://localhost:8080`. You can test it using a tool like Postman or cURL:
	
	- **GET request**: `GET http://localhost:8080/products`
	- **POST request**: `POST http://localhost:8080/products` with a JSON body like:
	```json
	{
	"id": 1,
	"name": "Sample Product",
	"price": 19.99
	}
	```
	
	That's it! You now have a simple REST API using Ktor to manage products.

Analysis
We used GPT and Gemini, using prompt chaining and zero shot prompting. We attempted to use prompt chaining to add the GET and POST requests separately, but GPT and Gemini both added GET and POST immediately. This was likely due to the comment in the given code saying to do both. Gemini provided little explanation of how it produced its code or how to use it. We decided that GPT with zero shot was the optimal combination due to its robust explanation and ease of prompting compared to prompt chaining. Our other results can be found on rows 18 and 19 of Problems_Explored.csv.

Name	Problem 10		
Goal	Properly summarize a given function		
Model	gpt-4o-mini	**Method**	Roleplay
Temperature	0.1	**Token Limit**	1024

Prompt	You are an expert software engineer who is one of the leading experts in the world. Write a brief summary of the following python method:
	def reverse_words (sentence) :
	return ’ ’. join (sentence . split () [:: -1])		

Output	The `reverse_words` method takes a string input called `sentence`, which consists of a sequence of words. It splits the sentence into individual words using the `split()` method, which creates a list of words. The slicing operation `[::-1]` reverses the order of the words in this list. Finally, the method combines these reversed words back into a single string using the `join()` method with a space `' '` as the separator. The result is a new string where the original words are presented in reverse order. For example, if the input is "Hello world", the output will be "world Hello".

Analysis

In this problem we explored the roleplay prompt strategy with high (.9) and low (.1) temperatures on the Gemini and GPT models to see if temperature made any significant difference in model's roleplay. The lower temperatures ended up doing a better job getting straight to the point, with gpt-4o-mini's low temperature output being the best. Our other results can be found on rows 20-21 of Problems_Explored.csv.

Name	Problem 11		
Goal	Write a prompt which could generate the following 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
```

Model	gpt-4o-mini	**Method**	Zero Shot
Temperature	0.7	**Token Limit**	1024
Prompt	Create a Python function is_prime that takes an integer n as input and returns True or False if n is prime or not.		
Output	Certainly! Here's a Python function `is_prime` that checks whether a given integer `n` is a prime number:		

~~~
```python
def is_prime(n):
    """Return True if n is a prime number, otherwise False."""
    if n <= 1:
~~~

~~~~
        return False  # 0 and 1 are not prime numbers
    if n <= 3:
        return True   # 2 and 3 are prime numbers
    if n % 2 == 0 or n % 3 == 0:
        return False  # eliminate multiples of 2 and 3

    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False  # eliminate multiples of 5 and 7, etc.
        i += 6

    return True  # n is prime

# Example usage
print(is_prime(11))  # Output: True
print(is_prime(4))   # Output: False
```

### Explanation:
1. **Initial Checks**:
   - Numbers less than or equal to 1 are not prime.
   - 2 and 3 are the first two prime numbers.

2. **Elimination of Even Numbers**:
   - If `n` is divisible by 2 or 3, it is not a prime.

3. **Checking for Factors**:
   - The loop starts at 5 and checks for factors up to the square root of `n` using the 6k +/- 1 rule. This means it checks for divisibility by all numbers of the form \(6k-1\) and \(6k+1\), which are the only forms that can be primes greater than 3.

Feel to test the function with different input values!
~~~~

Analysis

We used GPT and Gemini with zero shot and CoT prompting. Interestingly, both GPT and Gemini used the optimal 6k +/- 1 rule to determine primacy when prompted with zero shot, but failed to produce this when prompted with CoT. Similarly to previous problems, Chain of Thought prompting resulted in some redundant information, whereas zero shot was more concise. GPT and zero shot created the optimal combination, being non redundant but still providing an easily-readable, robust explanation of its code. Our other outputs can be found in rows 22 and 23 of Problems_Explored.csv.

Name	Problem 12		
Goal	Fix the factorial bug when input = 0		
Model	Ministral-3B	**Method**	Chain of Thought
Temperature	0.7	**Token Limit**	1024

Prompt

Identify and fix the bug with the following factorial function when the input = 0

```
def factorial ( n ) :
result = 1
for i in range (1 , n ) :
result *= i
return result
```

Output

The bug in the provided factorial function is in the range argument of the `for` loop. The range should include the `n` value itself, so the loop should iterate from `1` to `n`. Additionally, the loop should be adjusted to handle the case when `n` is 0, which should return 1 as per the definition of the factorial function for non-negative integers.

Here is the corrected version of the factorial function:

```python
def factorial(n):
    if n == 0:
    return 1
    result = 1
    for i in range(1, n + 1):
    result *= i
    return result
```

### Explanation:
1. **Handling `n = 0` Case**: Added a check at the beginning to return 1 if `n` is 0.
2. **Correcting the Range**: Changed the range in the `for` loop to `range(1, n + 1)` to include `n` in the iteration.

With these changes, the function will correctly compute the factorial for any non-negative integer `n`, including `n = 0`.

Analysis

In problem 12 we compared prompt chaining vs chain-of-thought prompting on Llama and Ministral. Llama-4-Scout-17B-16E-Instruct seems to like to ramble on and repeat information, letting Ministral take the lead with its more concise answers to the prompt. Overall, giving each step to the AI in a single prompt leads to a better response. Perhaps this is due to the nature of AI doing better with more context, instead of feeding the AI a step at a time. Our other results for this problem can be found on rows 24-25 of Problems_Explored.csv.

Name	Problem 13		
Goal	Implement by-value node deletion in a C linked list		
Model	Gemini	**Method**	Zero Shot
Temperature	0.7	**Token Limit**	1024
Prompt	You are a computer science researcher seeking to optimize linked lists. Implement node deletion by value in this C linked list:		
Output			

~~~~
```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data ;
  struct Node * next ;
};

void deleteNode ( struct Node ** head , int key ) {
  // Store head node
  struct Node* temp = *head, *prev;

  // If head node itself holds the key to be deleted
  if (temp != NULL && temp->data == key) {
    *head = temp->next;   // Changed head
    free(temp);           // free old head
    return;
  }

  // Search for the key to be deleted, keep track of the
  // previous node as we need to change 'prev->next'
  while (temp != NULL && temp->data != key) {
    prev = temp;
    temp = temp->next;
  }
~~~~

```
  // If key was not present in linked list
  if (temp == NULL) return;

  // Unlink the node from linked list
  prev->next = temp->next;

  // Free memory
  free(temp);
}

// A utility function to print linked list
void printList(struct Node *node) {
  while (node != NULL) {
    printf(" %d ", node->data);
    node = node->next;
  }
  printf("\n");
}

// Driver program to test above function
int main() {
  struct Node* head = NULL;
  struct Node* second = NULL;
  struct Node* third = NULL;

  // allocate 3 nodes in the heap
  head  = (struct Node*)malloc(sizeof(struct Node));
  second = (struct Node*)malloc(sizeof(struct Node));
  third = (struct Node*)malloc(sizeof(struct Node));

  head->data = 1; //assign data in first node
  head->next = second; // Link first node with the second node

  second->data = 2; //assign data to second node
  second->next = third;

  third->data = 3; //assign data to third node
  third->next = NULL;

  printf("Created Linked List: ");
  printList(head);

  deleteNode(&head, 2); // Delete node with data 2

  printf("Linked List after Deletion of 2: ");
  printList(head);
  deleteNode(&head, 1); // Delete node with data 1

  printf("Linked List after Deletion of 1: ");
  printList(head);

  deleteNode(&head, 3); // Delete node with data 3

  printf("Linked List after Deletion of 3: ");
  printList(head);

  return 0;
}
```

Key improvements and explanations:

* **Handles the head node case:** The code now correctly handles the case where the node to be deleted is the head node. It updates the `head` pointer directly. This was the most common error in previous versions.
* **Handles the case where the key is not found:** The code includes a check to ensure that the key exists in the linked list. If the key is not found, the function returns without making any changes. This prevents unexpected behavior.
* **Correctly updates `prev->next`:** The code accurately updates the `next` pointer of the previous node to bypass the node being deleted. This ensures that the linked list remains intact after deletion.
* **Frees memory:** The code frees the memory occupied by the deleted node using `free(temp)`. This prevents memory leaks.
* **Clearer variable names:** Uses more descriptive variable names like `temp` and `prev` to improve readability.
* **Complete, runnable example:** Includes a `main` function that demonstrates how to use the `deleteNode` function and includes a `printList` function for verification. This makes it easy to test the code.
* **Includes necessary header files:** Includes `stdio.h` for `printf` and `stdlib.h` for `malloc` and `free`.
* **Comments:** Added comments to explain the different parts of the code.
* **Safe handling of `NULL` head:** The code checks if the head pointer is `NULL` before attempting to dereference it. This prevents segmentation faults if the linked list is empty.

How to compile and run:

1. **Save:** Save the code as a `.c` file (e.g., `delete_node.c`).
2. **Compile:** Open a terminal or command prompt and compile the code

	using a C compiler (like GCC):

~~~
    ```bash
    gcc delete_node.
~~~

Analysis
We used GPT and Gemini, using zero shot and roleplay prompting. Roleplay seemed to have little to no effect on the outputs except for Gemini claiming that its entirely normal implementation of deletion was an “improvement.” All 4 combinations again produced entirely acceptable output, although the Gemini-roleplay prompt hit its token limit of 1024. We determined Gemini and zero shot to be the optimal combination, but all combinations were satisfactory. The other outputs can be found on rows 26 and 27 of Problems_Explored.csv.

Name	Problem 14		
Goal	Complete the recursive Fibonacci function		
Model	Ministral-3B	**Method**	Zero Shot w/ low temp
Temperature	0.1	**Token Limit**	1024
Prompt	Given the following incomplete Python method, build a recursive function for Fibonacci: def fibonacci (n) : # TODO : Base cases and recursive call pass		
Output	Certainly! The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. Here's how you can build a recursive function for Fibonacci in Python:		

~~~
```python
def fibonacci(n):
        # Base cases
        if n <= 0:
        return 0
        elif n == 1:
        return 1
        # Recursive case
        else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Example usage:
~~~

	print(fibonacci(10)) # Output: 55
	```
	### Explanation:
	1. Base Cases:
	   - If `n` is 0, return 0.
	   - If `n` is 1, return 1.

	2. Recursive Case:
	   - For any other value of `n`, return the sum of the Fibonacci numbers at `n-1` and `n-2`.

	### Example:
	- `fibonacci(10)` will compute `fibonacci(9) + fibonacci(8)`, which in turn will compute `fibonacci(8) + fibonacci(7)`, and so on, until it reaches the base cases.

	This recursive approach is simple but not the most efficient for large values of `n` due to its exponential time complexity. For larger values, consider using memoization or an iterative approach to improve performance.
	```

Analysis
In problem 14 we compared the effects of low and high temperature on zero shot prompting for a simple coding problem (complete the fibonacci function). The LLMs tended to ramble less with low temp prompts rather, and were more likely to go tangents at a high temperature. Ministral again proved superior to Llama, and less likely overall to go on tangents. While the step by step breakdown of Llama's outputs would be nice for beginning coders, the rambling would get old fast. Our other results for this problem can be found on rows 28-29 in Problems_Explored.csv.

Name	Problem 15		
Goal	Complete an unfinished Python constructor		
Model	gpt-4o-mini	**Method**	Zero Shot
Temperature	0.7	**Token Limit**	1024

Prompt

Add name, age, and optional email variables to the constructor of the Person class.

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

Output

You can modify the `Person` class constructor to include `name`, `age`, and an optional `email` parameter. Here's how you can do it:

~~~
```python
class Person:
    def __init__(self, name, age, email=None):
        self.name = name
        self.age = age
        self.email = email

# Example usage:
person1 ="Alice", 30)
person2 = Person("Bob", 25, "bob@example.com")

print(person1.name, person1.age, person1.email)  # Output: Alice 30 None
print(person2.name, person2.age, person2.email)  # Output: Bob 25
bob@example.com
```
~~~

In this implementation:
- `name` and `age` are required parameters.
- `email` is an optional parameter with a default value of `None`. This means that if the caller does not provide an email, it will simply be set to `None`.

Analysis

We used GPT and Gemini, using prompt chaining and zero shot prompts. We attempted to build the constructor in 2 steps: adding name and age, and adding email. However, likely due to the comment saying to add all 3, the first prompts in our chains for both GPT and Gemini built the entire constructor. All 4 combinations built the code correctly. GPT and zero shot were chosen as the optimal combination because of its conciseness, but all four outputs were very similar. The others can be found in rows 30 and 31 in Problems_Explored.csv.

Name	Problem 16		
Goal	Complete the binary search implementation (self-consistency vs zero-shot		
Model	Llama-4-Scout-17B-16E-Instruct	**Method**	zero-shot
Temperature	0.7	**Token Limit**	1024
Prompt	Given the following code, complete the binary search implementation. Do not give an explanation		

```
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;
}
```

Output

~~~
```java
public int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
        int mid = (left + right) / 2;
        // Compare and adjust bounds
        if (arr[mid] == target) {
        return mid;
        } else if (arr[mid] < target) {
        left = mid + 1;
        } else {
        right = mid - 1;
        }
        }
~~~

Analysis

In problem 16 we compared the zero-shot prompt strategy with self-consistency at the regular temperature. Both Ministral and Llama gave acceptable results in zero-shot and self-consistency, giving practically the same output every time. Llama had a slight edge, leaving comments on the code it created without being asked, allowing for better understanding of the code. Overall, self-consistency did not seem to do too much to change the results of LLMs. Our other results for this problem can be found on rows 32-33 in Problems_Explored.csv.

Name	Problem 17		
Goal	Resolve the inconsistency between a function's name and actual action		
Model	Gemini	**Method**	Chain of Thought
Temperature	0.2	**Token Limit**	1024

Prompt	Resolve the inconsistency between this C++ function name and its logic.
	Explain each step of your process.
	// Supposed to return true if x is even
	bool isOdd (int x) {		

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

Output

Here are a few ways to correct the function:

Option 1: Correct the Logic (Preferred - Simplest and Most Readable)

```c++
// Returns true if x is odd
bool isOdd (int x ) {
  return x % 2 != 0; // Returns true if x is odd
}
```

Explanation:

*   `x % 2 != 0`: This expression checks if the remainder of `x` divided by 2 is *not* equal to 0. If the remainder is not 0 (i.e., it's 1), then `x` is odd, and the expression evaluates to `true`.

Option 2: Rename the Function (Less Preferred - Can be confusing if other code relies on the name)

```c++
// Returns true if x is even
bool isEven (int x ) {
  return x % 2 == 0; // Returns true if x is even
}
```

Explanation:

*   This option keeps the original logic but renames the function to `isEven` to accurately reflect what the function does. However, this is generally less desirable because other parts of the code might be expecting a function called `isOdd`. Changing the name could break existing code.

Option 3: Use the `!` (Not) Operator (Less Readable, Avoid if possible)

```c++
// Returns true if x is odd
bool isOdd (int x ) {
  return !(x % 2 == 0); // Returns true if x is odd
}
```

Explanation:

*   This option uses the `!` (logical NOT) operator to invert the result of the original expression. If `x % 2 == 0` is `true` (meaning `x` is even), the `!` operator makes it `false`. If `x % 2 == 0` is `false` (meaning `x` is odd), the `!` operator makes it `true`.
*   While this works, it's generally considered less readable than directly checking `x % 2 != 0`. It forces the reader to do an extra mental step to understand the logic.

4. Justification for Choosing Option 1:

*   **Clarity:** `x % 2 != 0` is the most direct and easily understood way to express "x is odd."
*   **Maintainability:** It's less likely to be misinterpreted or require further explanation in the future.
*   **Least Disruptive:** It keeps the function name consistent with its intended purpose, minimizing the risk of breaking existing code that relies on the name `isOdd`.

Final Answer (Option 1):

```c++
// Returns true if x is odd
bool isOdd (int x ) {
  return x % 2 != 0; // Returns true if x is odd
}
```

Analysis

We used GPT and Gemini, using CoT prompting with different temperatures. With both GPT and Gemini, higher temperatures resulted in excessively wordy responses that were more confusing than their low-temperature counterparts. To resolve the discrepancy between function name and logic, Gemini low-temp proved optimal. It listed different options for how to resolve the discrepancy and explained genuine merits or flaws for each. It also, correctly, chose to change the function’s logic instead of its name. The other outputs can be found in rows 34 and 35 in Problems_Explored.csv.

Name	Problem 18
Goal	Identify and fix a JavaScript bug

Model	Ministral-3B	**Method**	Zero Shot
Temperature	0.5	**Token Limit**	1024

Prompt

Identify the bug in the following Javascript function, then fix the identified bug. Briefly explain your reasoning.

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

Output

The bug in the given JavaScript function is that it returns the remainder of the division of `n` by 2 (`n % 2`), which is either 0 or 1. However, the function is intended to return a boolean value (`true` or `false`) indicating whether the number is even or odd.

To fix this bug, we need to modify the function to return `true` if `n % 2` is 0 (indicating that `n` is even) and `false` otherwise.

Here is the corrected function:

```javascript
function isEven(n) {
   return n % 2 === 0; // Returns true if n is even, false otherwise
}
```

### Explanation:
- The original function `return n % 2;` returns either 0 or 1, which is not a boolean value.
- To fix this, we use the equality operator `===` to check if `n % 2` is exactly 0. If it is, the function returns `true`, indicating that `n` is even. Otherwise, it returns `false`, indicating that `n` is odd.

This change ensures that the function behaves as expected, returning a boolean value based on whether the input number is even or odd.

Analysis

We compared Llama and Ministral on zero shot prompts with high and low temperatures. At the moderate temperature of 0.5, both models performed well, but at 1.5, Llama was less concise and Ministral erupted into a stream of nonsensical tokens. We determined Ministral at temperature 0.5 to be the optimal combination for this problem, as it was highly concise and explained its code effectively. Llama performed similarly, but less concisely, like a chain of thought prompt. The other outputs can be found in rows 36 and 37 of Problems_Explored.csv.

Name	Problem 19		
Goal	Create a step-by-step summary of given C++ code		
Model	gpt-4o-mini	**Method**	Prompt Chaining
Temperature	0.7	**Token Limit**	1024

Prompt

First prompt: Create a summary of this C++ function:

```
// Function that validates an input , calculates square , and returns result
int process (int x ) {
  if ( x < 0) return -1;
  return x * x ;
})
```

Second prompt: Decompose that summary into logical steps.

Output

Assistant: The provided C++ function, `process`, takes an integer input `x` and performs the following actions:

1. It checks if `x` is less than 0. If it is, the function returns -1, indicating that the input is invalid.
2. If `x` is non-negative, the function calculates the square of `x` (i.e., `x * x`) and returns the result.

In summary, this function validates the input and returns the square of the input if it is non-negative; otherwise, it returns -1 for invalid input.
Assistant: Here’s a decomposition of the summary into logical steps:

1. **Function Definition**:
   - The function is named `process` and takes a single integer parameter `x`.

2. **Input Validation**:
   - The function checks if the input `x` is less than 0.
     - If `x < 0`, it indicates that the input is invalid.

3. **Return for Invalid Input**:
   - If the input is invalid (i.e., `x < 0`), the function returns -1.

4. **Calculate Square for Valid Input**:
   - If the input is valid (i.e., `x` is non-negative), the functions to calculate the square of `x` using the expression `x * x`.

5. **Return Result**:
   - The function returns the calculated square of `x` as the result.

This breakdown outlines the logical flow of the function from input

	validation to the final output.

Analysis
We used GPT and Gemini, using zero shot and prompt chaining strategies. The zero-shot outputs for both GPT and Gemini provided a reasonable step-by-step summary, but the GPT prompt chaining was superior, as it demonstrated a valuable ability of the model: it provided a more concise summary, then went into further detail as necessary. The zero shot outputs provided what would likely be excessive detail for many users, whereas the prompt chaining allowed the user to control the amount of explanation they wanted. The Gemini prompt chaining was also highly successful, but provided more redundant information than GPT, thus leaving the combination of GPT and prompt chaining as optimal. The other outputs can be found in rows 38 and 39 in Problems_Explored.csv.

Name	Problem 20		
Goal	Complete an unfinished Python function		
Model	Ministral-3B	**Method**	Chain of Thought
Temperature	0.7	**Token Limit**	1024
Prompt	Complete the following python method based on the intent of the function. Explain your reasoning. Do this step by step. def calculate_average (scores) : total = 0 # TODO : Complete to return average pass		
Output	To complete the `calculate_average` method, we need to follow these steps: 1. **Sum up all the scores**: Initialize a variable `total` to store the sum of the scores. 2. **Calculate the average**: Divide the `total` by the number of scores to get the average. 3. **Return the average**: Return the calculated average. Let's go through each step: ### Step 1: Sum up all the scores We already have the `total` variable initialized to 0. We need to iterate over the list of scores and add each score to `total`.		

~~~
### Step 2: Calculate the average

Once we have the total sum of scores, we can calculate the average by dividing `total` by the number of scores.

### Step 3: Return the average

Finally, we return the calculated average.

Here's the complete function:

```python
def calculate_average(scores):
    total = 0
    for score in scores:
        total += score
    average = total / len(scores)
    return average
```

### Explanation:

1. **Initialization**: `total = 0` ensures that we start with a clean total.
2. **Summation**:
   - `for score in scores:` iterates over each score in the `scores` list.
   - `total += score` adds the current score to `total`.
3. **Calculating the average**:
   - `average = total / len(scores)` divides the sum of scores by the number of scores to get the average.
4. **Return**: `return average` returns the calculated average.

### Example Usage:

```python
scores = [85, 90, 78, 92]
print(calculate_average(scores))  # Output: 86.5
```


This method will correctly calculate the average of the list of scores provided.
~~~

Analysis

We used Llama and Ministral, using CoT with different temperatures. Both models provided entirely acceptable outputs at temperature 0.7, but at a searing temperature of 1.5, Llama's response became messier and Ministral completely collapsed. We determined Ministral at temperature 0.7 to be the optimal combination, as it was slightly more concise than Llama, but both explained their code well. Our other outputs can be found in rows 40 and 41 of Problems_Explored.csv.

4 Results

In many cases, a simple zero shot prompt was the most effective way to get concise answers from all AI models. However, for more complex topics or for ones where the user needs more guidance, strategies like chain of thought and prompt chaining were more appropriate. We found roleplaying to be limited in its effectiveness, as the role we wanted the AI to play was invariably very similar to its normal role: a highly knowledgeable assistant. Differences between the models were slight, but present. GPT 4o mini was highly reliable and commonly our preferred choice. However, it sometimes was overly verbose in response to simple prompts. Codestral-2501 did a fine job, but was outclassed by GPT 4o mini much of the time, giving less concise results. Llama-4-Scout-17B-16E-Instruct was overall good at staying focused and on task, even at higher temperatures, but has a tendency to go on rambles, and restate redundant information. Ministral-3B was reasonably concise and effective, though it collapsed at extreme temperatures. Gemini-2.0-flash tended to be more concise, sometimes to a fault as in Problem 1. All models produced highly reliable solutions to the problems presented to them and rarely needed any kind of correction. Increasing temperature could cause them to behave wildly, but only when increased to extreme levels like 1.5. Some also had common phrases, such as Gemini's tendency to start outputs with the phrase, "Ok, let's…" and GPT and Codestral's "Certainly!" Any of these models utterly trivializes simple coding challenges.