1. **Code Summarization (Java)** Summarize the functionality of the following method:

```java
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;
}
```

Selected Models: ChatGPT and Claude

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Summarize the functionality of the following method: (method)
ChatGPT Output:

Summary:
The `countWordFrequency` method takes a list of strings (`words`) and returns a map that contains each unique word as a key and the number of times it appears in the list as the value. In other words, it computes and returns the frequency count of each word in the input list.

Claude Output:

This method `countWordFrequency` takes a list of strings (`words`) as input and returns a map that counts how many times each unique string appears in the list.

Here's what it does step by step:

1. Creates an empty HashMap called `freqMap` to store word frequencies
2. Iterates through each string in the input list
3. For each word, adds it to the map or updates its count:
   - If the word is not yet in the map, it puts it with a count of 1
   - If the word is already in the map, it increments its count by 1
4. Returns the completed frequency map where keys are the unique words and values are their occurrence counts

This is a common pattern for counting frequencies of elements in a collection.

Comment: Claude is much more thorough than ChatGPT on this zero-shot prompting. It explains the code as if I have barely coded before and gets into the technical aspects. ChatGPT is very straight to the point.

Prompting Strategy 2:
Prompt: Explain in a step by step process your thoughts before coming to a final answer. Summarize the functionality of the following method: (method)

ChatGPT Output:

Step 1: Read the Method Signature

```java
public Map < String , Integer > countWordFrequency ( List < String > words )
```

- The method is public.
- It returns a `Map<String, Integer>`.
- It takes as input a `List<String>` called `words`.

Step 2: Examine the First Statement

```java
Map < String , Integer > freqMap = new HashMap < >() ;
```

- A new empty `HashMap` is created to store results.
- The map will map strings (words) to integers (counts).

Step 3: Analyze the For Loop

```java
for ( String word : words ) {
    freqMap . put ( word , freqMap . getOrDefault ( word , 0) + 1) ;
}
```

- The loop iterates over each word in the input list.
- For each word:
  - `freqMap.getOrDefault(word, 0)` retrieves the current count for the word, or 0 if the word hasn't been seen yet.
  - It adds 1 to this count.
  - It puts the updated count back into the map for that word.
- This effectively counts how many times each word appears in the list.

Step 4: Return Statement

```java
return freqMap ;
```

- The map containing the word counts is returned.

Step 5: Synthesize the Steps

- The method creates a map.
- It counts occurrences of each word in the input list.
- It returns a map of words to their frequencies.

Final Summary

The `countWordFrequency` method takes a list of words and returns a map where each key is a unique word from the list and its value is the number of times that word appears in the list. It computes the frequency of each word in the input list.

Claude Output:

Here's my step-by-step thought process for analyzing this Java method:

Step 1: First, I'll examine the method signature to understand inputs and outputs.

- The method is called `countWordFrequency`
- It takes a parameter `List<String> words` (a list of strings)
- It returns a `Map<String, Integer>` (mapping from strings to integers)
- Based on the name and types, this likely counts occurrences of words

Step 2: Next, I'll analyze the first line of code inside the method.

- `Map<String, Integer> freqMap = new HashMap<>();`
- This creates an empty HashMap to store results
- Keys will be strings (words), values will be integers (counts)

Step 3: Now I'll analyze the loop structure.

- `for (String word : words)` iterates through each string in the input list
- This confirms we're processing each word from the input list

Step 4: Let's understand the core logic inside the loop.

- `freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);`
- This line does multiple things:
  - `freqMap.getOrDefault(word, 0)` checks if the word exists in the map
  - If it exists, retrieves its current count; if not, returns 0
  - Adds 1 to this count (incrementing the frequency)
  - `freqMap.put(word, ...)` then updates the map with this new count

Step 5: Finally, analyzing the last line.

- `return freqMap;` returns the completed frequency map to the caller

Comment: This prompting strategy is definitely a little bit excessive for a problem this simple. Both models give much longer explanations that feel like they are explaining the basic function of how the language Java works rather than giving a simple answer. Might be helpful for someone with little experience coding.

2. **Bug Fixing (Python – Off-by-One)** Identify and fix the off-by-one error in this function:

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

Selected Models: Gemini and LLaMA

Prompting Strategy 1: Three-Shot Prompting

Prompt:

Bug Fixing (Python – Off-by-One)
Example 1:
Identify and fix the off-by-one error in this function:
Python

```
def count_inclusive(start, end):
    count = 0
    for i in range(start, end):
        count += 1
    return count
```

Solution:
The range should include `end`, so use `range(start, end + 1)`.

Example 2:
Identify and fix the off-by-one error in this function:
Python

```
def create_list(n):
    result = []
    for i in range(n):
        result.append(i)
    return result
```

Solution:
No off-by-one error here; the function correctly creates a list from 0 to n-1.

Example 3:
Identify and fix the off-by-one error in this function:
Python

```
def sum_first_n(n):
    total = 0
    for i in range(1, n):
        total += i
    return total
```

Solution:
To include `n` in the sum, use `range(1, n + 1)`.

Now, try this one:
Bug Fixing (Python – Off-by-One) Identify and fix the off-by-one error in this function:
Python

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

Gemini Output:

Python

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

Solution:

The `range` function in Python excludes the upper bound. To include the `end` value in the summation, the `range` should be `range(start, end + 1)`. This ensures that the loop iterates through all numbers from `start` up to and including `end`.

LLaMA Output:

The off-by-one error in this function is that the range should include `end`, but the current implementation only goes up to `end - 1`. To fix this, we need to change the `range` function 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 total
```

Comment:

Prompting Strategy 2: Zero-Shot Prompting
Prompt: Identify and fix the off-by-one error in this function:
Python

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

Gemini Output:

You've spotted a classic off-by-one error! The `range(start, end)` function in Python includes the `start` value but *excludes* the `end` value. So, your function as it is will calculate the sum up to, but not including, the `end` number.

To fix this, you need to adjust the `range` to include the `end` value. Here's the corrected function:

Python

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

By changing `range(start, end)` to `range(start, end + 1)`, you ensure that the loop iterates through all the numbers from `start` up to and including `end`.

LLaMA Output:

Here is the corrected function:

Python

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

Alternatively, you can use the `sum` function with the `range` function to make the code more concise:

Python

```
def sum_range(start, end):
    return sum(range(start, end + 1))
```

However, there's an even simpler way to fix this function:

Python

```
def sum_range(start, end):
    return sum(range(start, end + 1))
```

can be simplified to:

Python

```
def sum_range(start, end):
    return sum(range(start, end + 1))
```

can be simplified to:

Python

```
def sum_range(start, end):
    return sum(range(start, end + 1))
```

can be simplified to:

Python

```
def sum_range(start, end):
    return sum(range(start, end + 1))
```

Comment: LLaMA first gave a pretty reasonable answer before getting stuck in an infinite loop. Both actually gave the same exact primary answer.

3. **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
}
```

Selected Models: LLaMA and ChatGPT

Prompting Strategy 1: Self Consistency prompting
Prompt: Classify the type of bug in the following C++ function: (function)

ChatGPT Output 1:

In summary:

The bug is a **dangling pointer** caused by returning the address of a local (stack-allocated) variable.

Correct Approach:

- Use `new` to allocate dynamic memory (caller must remember to `delete[]` it), or
- Use `std::vector<int>` and return a vector by value (preferred in modern C++).

ChatGPT Output 2:

Classification of the Bug:

- **Type:** Returning pointer to local (stack) variable
- **Category:** Lifetime error / Dangling pointer

Explanation:

- `arr` is a local variable allocated on the stack inside the function.
- When the function returns, the memory for `arr` is deallocated.
- Returning a pointer to this memory (`arr`) means the caller will receive a pointer to memory that is no longer valid (dangling pointer).
- Using this pointer leads to **undefined behavior**.

ChatGPT Output 3:

Type of Bug:
Returning pointer to a local (stack-allocated) variable

Detailed Explanation:

- The array `arr` is declared inside the function, so it has **automatic (stack) storage duration**.
- When the function returns, `arr` goes out of scope and **ceases to exist**.
- Returning `arr` (as a pointer) means returning a pointer to memory that is **no longer valid** after the function exits.
- Dereferencing this pointer in the caller leads to **undefined behavior**.

How to Fix:

- Allocate the array on the heap using `new int[size]` , or
- Use `std::vector<int>` and return the vector.

LLaMA Output 1:

Type of Bug:

This bug is classified as a **Memory Leak** and a **Dangling Pointer** issue. The memory leak occurs because the function allocates memory on the stack that is not properly released, and the dangling pointer issue occurs because the returned pointer points to memory that is no longer valid.

How to Fix:

To fix this bug, the function should allocate memory on the heap using `new` or `malloc` and return a pointer to that memory. Alternatively, the function can return a reference to a local array, but this would require changing the function signature to `int& getArray(int size)`.

Here's an example of how to fix the bug using `new`:

LLaMA Output 2:

Bug Classification: Use-After-Free

The bug in the given C++ function is a classic example of a Use-After-Free error.

What is a Use-After-Free error?

A Use-After-Free error occurs when a program attempts to access memory that has already been freed or deallocated. In this case, the function `getArray` returns a pointer to a local array, which is automatically destroyed when the function returns. The returned pointer becomes a dangling pointer, pointing to memory that is no longer valid.

How to fix this bug?

To fix this bug, the function should allocate memory on the heap using `new` and return a pointer to that memory. The caller must then be responsible for deallocating the memory using `delete`. Here's the corrected function:

C++

```
int* getArray(int size) {
    int* arr = new int[size];
    return arr;
}
```

LLaMA Output 3:

The bug in this C++ function is an example of a **UB (Undefined Behavior) due to Dangling Pointer**.

Recommended Fix:

To fix this bug, the function should allocate the array on the heap using `new[]` and return a pointer to the first element. Alternatively, the function can return a copy of the array or a smart pointer to manage the array's lifetime.

Comment: ChatGPT had a relatively consistent output, but LLaMA had some variance in its bug classification. It also seemed that LLaMA would try to provide much lengthier responses on average than ChatGPT.

Prompting Strategy 2: Chain of thought
Prompt: Explain in a step by step process your thoughts before coming to a final answer. Classify the type of bug in the following C++ function: (function)

ChatGPT Output:

Step 1: Read and Understand the Code

C++

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

This function takes an integer `size` and declares a local array `arr` of that size, then returns a pointer to that array.

Step 2: Identify the Storage Duration of `arr`

- `arr` is declared inside the function with no `static` or `new` keyword.
- This means `arr` is a **local variable** with **automatic storage duration**.
- Its memory is allocated on the **stack** and is only valid inside the function.

Step 3: What Happens When the Function Returns?

- When the function returns, any local (automatic) variables are **destroyed**.
- The memory for `arr` is **reclaimed**.
- Returning `arr` actually returns the **address of memory that is no longer valid**.

Step 4: What is the Consequence?

- The returned pointer points to memory that **could be overwritten or used for other purposes** by the program.
- Accessing this pointer leads to **undefined behavior**: crashes, wrong data, or subtle bugs.

Step 5: Is This a Known Bug Pattern?

- Yes, returning a pointer (or reference) to a local variable is a **classic C++ bug**.
- This is commonly called a **dangling pointer**.
- Other terms: **use-after-scope**, **returning address of local variable**, **lifetime issue**.

Step 6: Final Classification

- The bug is a **dangling pointer** — returning a pointer to a local variable that no longer exists.
- This is also a **lifetime issue**.

Final Answer

Type of bug:
Dangling pointer (returning pointer to a local variable / lifetime issue / use-after-scope)

Explanation:
Returning a pointer to a local (stack-allocated) variable results in a dangling pointer, which leads to undefined behavior when accessed after the function returns.

LLaMA Output:

Step 1: Analyze the function signature and return type

- The function `getArray` takes an `int` parameter `size` and returns a pointer to an `int`, `int *`.
- The return type is a pointer, which suggests that the function will allocate memory for the returned value on the heap.

Step 2: Examine the local variable declaration

- The function declares a local array `arr` of size `size` using the syntax `int arr [ size ];`.
- This is a warning in C++, as the size of the array is not known at compile-time.
- However, this does not immediately indicate a bug.

Step 3: Check the return statement

- The function returns the address of the local array `arr` using the expression `return arr;`.
- This is where the bug becomes apparent.

Step 4: Consider the lifetime of the local variable

- The local array `arr` is allocated on the stack and has a limited lifetime, which is tied to the execution of the function.
- Once the function returns, the local array `arr` is destroyed, and its memory is deallocated.

Step 5: Analyze the returned pointer

- The returned pointer points to the memory location of the local array `arr`.
- However, since the local array has been destroyed, the returned pointer is now pointing to invalid memory.

Step 6: Identify the type of bug

- Based on the analysis, the bug is a case of **returning a pointer to a local variable**, which is a form of **dangling pointer**.
- This is a type of **memory-related bug**, specifically a **scope and lifetime bug**.

Final Answer:

The type of bug in the given C++ function is a **dangling pointer** or **returning a pointer to a local variable** bug, which is a form of **memory-related bug** and **scope and lifetime bug**.

Comment: LLaMA dives deeper into the actual function of the code when run, while ChatGPT identifies the problem and quickly classifies the type of bug.

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

```
def is_valid_email(email):
    # TODO: Complete using regex
    pass
```

Selected Models: Claude and Gemini

Prompting Strategy 1:
Prompt: Complete the function using regex to validate basic email addresses: (function)
Claude Output:

```
def is_valid_email(email):
    import re

    # Basic email validation pattern
    # Checks for:
    # - One or more characters before the @ symbol
    # - One or more characters followed by a dot after the @ symbol
    # - At least two characters after the last dot
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    if re.match(pattern, email):
        return True
    else:
        return False
```

Gemini Output:

```
import re

def is_valid_email(email):
    """
    Validates basic email addresses using regular expressions.

    Args:
        email: The email address string to validate.

    Returns:
        True if the email address matches the basic pattern, False otherwise.
    """
    # Basic pattern: one or more alphanumeric characters, periods, underscores,
    # or hyphens, followed by an "@" symbol, followed by one or more alphanumeric
    # characters or hyphens, followed by a period, followed by one or more
    # alphabetic characters.
    pattern = r"^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z]+$"
    return bool(re.match(pattern, email))

# Example usage:
print(is_valid_email("test@example.com"))       # Output: True
print(is_valid_email("user.name@sub.domain.com")) # Output: True
print(is_valid_email("user_123@test-site.org"))  # Output: True
print(is_valid_email("invalid_email"))          # Output: False
```

```
print(is_valid_email("@missing_local.com"))     # Output: False
print(is_valid_email("no_domain@.com"))         # Output: False
print(is_valid_email("no_tld@example."))        # Output: False
print(is_valid_email("space in@email.com"))     # Output: False
```

Comment: Both of these solutions explain and create a regular expression pattern and use the built in match function to match with the input. Gemini has slightly more thorough commenting and also provides example usage. Gemini requires only one character after the last dot, while Claude requires at least 2.

Prompting Strategy 2: Self-Consistency
Prompt: Complete the function using regex to validate basic email addresses: (function)
Claude Output 1:

Claude Output 2:

```
def is_valid_email(email):
    import re

    # Basic email pattern: username@domain.extension
    # Username: can contain letters, numbers, underscores, periods, plus signs, hyphens
    # Domain: can contain letters, numbers, periods, hyphens
    # Extension: at least 2 characters

    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'

    if re.match(pattern, email):
        return True
    return False
```

Claude Output 3:

```
def is_valid_email(email):
    import re

    # Basic email pattern: username@domain.extension
    # Username: alphanumeric characters, dots, underscores, hyphens, plus signs
    # Domain: alphanumeric characters, dots, hyphens
    # Extension: 2 or more alphabetic characters

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

    if re.match(pattern, email):
        return True
    else:
        return False
```

Gemini Output 1:

```
import re

def is_valid_email(email):
    """
    Validates basic email addresses using regular expressions.

    Args:
        email: The email address string to validate.

    Returns:
        True if the email address matches the basic pattern, False otherwise.
    """
    # Basic pattern: one or more alphanumeric characters, periods, underscores,
    # or hyphens, followed by an "@" symbol, followed by one or more alphanumeric
    # characters or hyphens, followed by a period, followed by one or more
    # alphabetic characters.
    pattern = r"^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z]+$"
    return bool(re.match(pattern, email))

# Example usage:
print(is_valid_email("test@example.com"))       # Output: True
print(is_valid_email("user.name@sub.domain.com")) # Output: True
print(is_valid_email("user_123@test-site.org"))  # Output: True
print(is_valid_email("invalid_email"))          # Output: False
print(is_valid_email("@missing_local.com"))     # Output: False
print(is_valid_email("no_domain@.com"))         # Output: False
print(is_valid_email("no_tld@example."))        # Output: False
print(is_valid_email("space in@email.com"))     # Output: False
```

Gemini Output 2:

```
import re

def is_valid_email(email):
    """
    Validates basic email addresses using regular expressions.

    Args:
        email: The email address string to validate.

    Returns:
        True if the email address matches the basic pattern, False otherwise.
    """
    # Basic pattern: one or more alphanumeric characters, dots, underscores,
```

```
    # or hyphens, followed by an "@" symbol, followed by one or more
    # alphanumeric characters or hyphens, followed by a dot, followed by
    # one or more alphanumeric characters.
    pattern = r"^[a-zA-Z0-9._-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9]+$"
    return bool(re.match(pattern, email))

# Example usage:
print(is_valid_email("test@example.com"))      # Output: True
print(is_valid_email("user.name@sub.domain.com")) # Output: True
print(is_valid_email("user_123@example-net.org")) # Output: True
print(is_valid_email("invalid email"))         # Output: False
print(is_valid_email("test@example"))          # Output: False
print(is_valid_email("@example.com"))          # Output: False
print(is_valid_email("test@.com"))             # Output: False
print(is_valid_email("test@example."))         # Output: False
print(is_valid_email("test@example.c"))        # Output: False
```

Gemini Output 3:

```
import re

def is_valid_email(email):
    """
    Validates basic email addresses using regular expressions.

    Args:
        email: A string representing the email address to validate.

    Returns:
        True if the email address matches the basic pattern, False otherwise.
    """
    # Basic pattern: one or more alphanumeric characters, periods, plus signs, or
underscores
    # followed by an "@" symbol, followed by one or more alphanumeric characters or
periods,
    # followed by a period ".", followed by two or more alphanumeric characters.
    pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    return bool(re.fullmatch(pattern, email))

# Example usage:
print(is_valid_email("test@example.com"))       # Output: True
print(is_valid_email("user.name+alias@sub.domain.co.uk")) # Output: True
print(is_valid_email("invalid-email"))          # Output: False
print(is_valid_email("@missing-local.com"))     # Output: False
print(is_valid_email("no-tld@example"))         # Output: False
```

print(is_valid_email(".leadingdot@example.com")) # Output: False
print(is_valid_email("trailingdot.@example.com"))# Output: False
print(is_valid_email("double..dot@example.com"))# Output: False

Comment: It seems that among both models the regular expression generation is inconsistent, and both need more work at it. The logic sometimes just doesn't track, even when the explanation is correct. These models are better at coding than regular expressions.

5. **Generating Flask APIs (Python)** Create a '/greet/<username>' endpoint that returns a JSON greeting:

```python
from flask import Flask, jsonify

app = Flask(__name__)

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

Selected Models: ChatGPT and Gemini

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Create a '/greet/' endpoint that returns a JSON greeting: (code)
ChatGPT Output:

Gemini Output:

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/greet/<username>')
def greet(username):
    """
    This function handles the '/greet/<username>' route.
    It takes a username as a path parameter and returns a JSON response
    containing a greeting message with the provided username.

    Args:
        username (str): The username to include in the greeting.

    Returns:
        json: A JSON object with a "message" key and a greeting string as the value.
              Returns a 200 status code.
    """
    greeting_message = f"Hello, {username}!"  # Construct the greeting string.
    response_data = {"message": greeting_message}  # Create a dictionary for the JSON response.
    return jsonify(response_data)  # Convert the dictionary to JSON and return it.

if __name__ == '__main__':
    app.run(debug=True) # Run the app in debug mode to see errors and changes without restarting.

```

Comment: Both output very similar code, Gemini just does a much more in-depth job explaining it with its comments.

Prompting Strategy 2:
Prompt: Create a '/greet/' endpoint that returns a JSON greeting: (code)
ChatGPT Output 1:

Here's a **step-by-step solution**:

1. Use Flask's `jsonify` function to return a dictionary as a JSON response.
2. Inside the view function, use the `username` parameter from the URL and insert it into the greeting string.

Python

```python
from flask import Flask, jsonify

app = Flask(__name__)

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

if __name__ == '__main__':
    app.run(debug=True)
```

How it works:

- Navigating to `/greet/Alice` will respond with

JSON

```
{
  "message": "Hello, Alice!"
}
```

- `jsonify` ensures the output is returned as proper JSON with correct headers.

ChatGPT Output 2:

1. Goal

You want a Flask endpoint `/greet/<username>` that returns a JSON greeting (e.g., `{"message": "Hello, username!"}`).

2. Approach

- Use Flask's `jsonify()` to return JSON.
- Retrieve the `username` parameter from the URL.
- Create a greeting message.

3. Implementation

Here's the solution:

Python

```python
from flask import Flask, jsonify

app = Flask(__name__)

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

# To run the app, uncomment the lines below:
# if __name__ == '__main__':
#     app.run(debug=True)
```

ChatGPT Output 3:

Absolutely! Here's how you can implement the `/greet/<username>` endpoint in Flask so it returns a JSON greeting:

Python

```python
from flask import Flask, jsonify

app = Flask(__name__)

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

# To run the app:
# if __name__ == '__main__':
#     app.run(debug=True)
```

How it works:

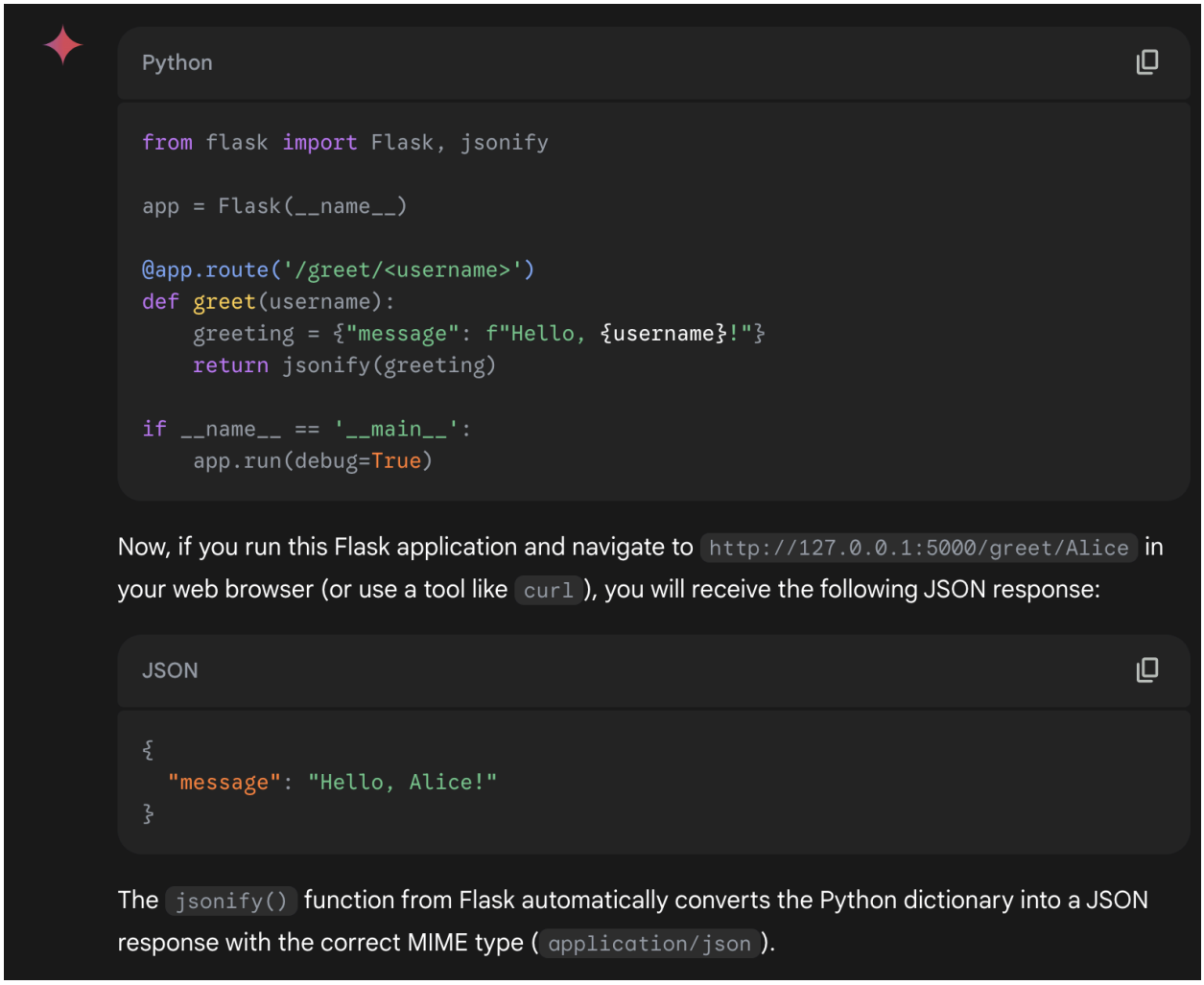
- When you visit `/greet/alice`, you'll get:
  `{"message":"Hello, alice!"}`

Let me know if you need more customization!

Gemini Output 1:

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/greet/<username>')
def greet(username):
    """
    This function handles the '/greet/<username>' route.
    It takes a username as a path parameter and returns a JSON response
    containing a greeting message with the provided username.

    Args:
        username (str): The username to include in the greeting.

    Returns:
        json: A JSON object with a "message" key and a greeting string as the value.
              Returns a 200 status code.
    """
    greeting_message = f"Hello, {username}!"  # Construct the greeting string.
    response_data = {"message": greeting_message}  # Create a dictionary for the JSON response.
    return jsonify(response_data)  # Convert the dictionary to JSON and return it.

if __name__ == '__main__':
    app.run(debug=True) # Run the app in debug mode to see errors and changes without restarting.

```

Gemini Output 2:



Python

```python
from 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)
```

Now, if you run this Flask application and navigate to `http://127.0.0.1:5000/greet/Alice` in your web browser (or use a tool like `curl`), you will receive the following JSON response:

JSON

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

The `jsonify()` function from Flask automatically converts the Python dictionary into a JSON response with the correct MIME type (`application/json`).

Gemini Output 3:
Comment: ChatGPT creates the same output consistently, but with different example names and with vs. without the main method commented out. Gemini only changes whether certain comments are added or not.

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

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

Selected Models: Claude and LLaMA

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Write the schema for a review app with users, books, and reviews: (Comments)
Claude Output:

```sql
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Books table
CREATE TABLE books (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    author TEXT NOT NULL,
    genre TEXT,
    publication_year INTEGER,
    isbn TEXT UNIQUE,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Reviews table
CREATE TABLE reviews (
    id INTEGER PRIMARY KEY,
    user_id INTEGER NOT NULL,
    book_id INTEGER NOT NULL,
    rating INTEGER NOT NULL CHECK (rating >= 1 AND rating <= 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,
    -- Ensure a user can only review a book once
    UNIQUE (user_id, book_id)
);

-- Index for faster lookup of reviews by user
CREATE INDEX idx_reviews_user_id ON reviews(user_id);

-- Index for faster lookup of reviews by book
CREATE INDEX idx_reviews_book_id ON reviews(book_id);
```

LLaMA Output:

Here is the schema for the review app:

SQL

```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 INTEGER NOT NULL REFERENCES users(id),
  book_id INTEGER NOT NULL REFERENCES books(id),
  rating INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5)
);
```

Let me explain the design choices:

Users table

- `id` : a unique identifier for each user, using the `SERIAL` data type to auto-increment.
- `name` : the user's name, with a maximum length of 255 characters.

Books table

- `id` : a unique identifier for each book, using the `SERIAL` data type to auto-increment.
- `title` : the book's title, with a maximum length of 255 characters.

Reviews table

- `id` : a unique identifier for each review, using the `SERIAL` data type to auto-increment.
- `user_id` : the foreign key referencing the `id` column in the `users` table, ensuring that each review is associated with a valid user.
- `book_id` : the foreign key referencing the `id` column in the `books` table, ensuring that each review is associated with a valid book.
- `rating` : the user's rating of the book, with a CHECK constraint to ensure that the rating is between 1 and 5 (inclusive).

By using foreign keys, we establish relationships between the tables, ensuring that:

- Each review is associated with a valid user and book.
- Each user can have multiple reviews, but each review is associated with only one user.
- Each book can have multiple reviews, but each review is associated with only one book.

This design allows for efficient querying and data integrity, while also supporting the core functionality of the review app.

Comment: Claude provides much more robust support for SQL tasks it seems upon first impression, while not in the screenshot, Claude also includes information about design decisions.

Prompting Strategy 2:

Prompt: Consider the following task and write a step-by-step analysis of what should go into completing it as best and robustly as possible. Then execute this vision. Write the schema for a review app with users, books, and reviews: (Comments)

Claude Output:

Step-by-Step Analysis for Creating a Robust Review App Schema

1. Understand the Core Requirements

First, let's understand what we need at minimum:

- Users table with id and name
- Books table with id and title
- Reviews table with id, user_id, book_id, and rating

2. Consider Additional Essential Fields

For each entity, we should consider what other fields would be essential:

Users:

- Email (for authentication and contact)
- Password storage (hashed)
- Registration date
- Last login timestamp

Books:

- Author
- Publication year
- ISBN (unique identifier for books)
- Genre
- Description

Reviews:

- Review text content (beyond just numerical rating)
- Timestamp of review creation
- Update timestamp

3. Consider Constraints and Rules

For Users:

- Email should be unique (no duplicate accounts)
- Name and email should be required (NOT NULL)
- Password must be stored securely (never in plaintext)

For Books:

- ISBN should be unique if provided
- Title must be required
- Consider if you want to enforce unique titles or allow duplicates

For Reviews:

- Rating should have a valid range (typically 1-5)
- A user should only be able to review a book once (unique constraint on user_id + book_id)
- Foreign key constraints to maintain referential integrity

4. Plan for Performance Optimization

We should add indexes for fields that will be frequently queried:

- User email (for login)
- Book title (for search)
- Reviews by user_id
- Reviews by book_id

5. Consider Data Integrity Rules

- What happens when a user is deleted? (Cascade or keep reviews?)
- What happens when a book is deleted?
- Should deleting reviews be allowed?

6. Plan for Future Expansion

Consider what features might be added later:

- User roles (admin, moderator)
- Book categories/tags
- Review helpful votes
- Comment threads on reviews

7. Create the Schema

After all considerations, I would create a schema that includes:

- Primary keys for each table
- Foreign keys with appropriate ON DELETE behavior
- CHECK constraints for data validation
- UNIQUE constraints where needed
- Default values where appropriate
- Timestamps for creation/updates

This thorough approach ensures the database will be robust, maintainable, and can grow with the application's needs without requiring significant restructuring later.

LLaMA Output:

Step 1: Identify the entities and their attributes

- **Users**: The users who will be writing reviews.
  - Attributes:
    - `id`: a unique identifier for each user.
    - `name`: the user's name.
    - `email`: the user's email address (optional).
    - `password`: the user's password (optional).
- **Books**: The books that will be reviewed.
  - Attributes:
    - `id`: a unique identifier for each book.
    - `title`: the book's title.
    - `author`: the book's author (optional).
    - `publisher`: the book's publisher (optional).
    - `publication_date`: the book's publication date (optional).
- **Reviews**: The reviews written by users for books.
  - Attributes:
    - `id`: a unique identifier for each review.
    - `user_id`: the foreign key referencing the `id` column in the `users` table.
    - `book_id`: the foreign key referencing the `id` column in the `books` table.
    - `rating`: the user's rating of the book (e.g., 1-5).
    - `review_text`: the user's review text (optional).

Step 2: Determine the relationships between entities

- A user can write multiple reviews, so there is a one-to-many relationship between `users` and `reviews`.
- A book can have multiple reviews, so there is a one-to-many relationship between `books` and `reviews`.
- A review is associated with one user and one book, so there is a many-to-one relationship between `reviews` and `users`, and a many-to-one relationship between `reviews` and `books`.

Step 3: Design the schema

Based on the entities, attributes, and relationships, the schema can be designed as follows:

SQL

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

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

CREATE TABLE reviews (
  id SERIAL PRIMARY KEY,
  user_id INTEGER NOT NULL REFERENCES users(id),
  book_id INTEGER NOT NULL REFERENCES books(id),
  rating INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5),
  review_text TEXT
);
```

Step 4: Add constraints and indexes

To ensure data integrity and improve query performance, the following constraints and indexes can be added:

- PRIMARY KEY constraints on the `id` columns of each table.
- FOREIGN KEY constraints on the `user_id` and `book_id` columns of the `reviews` table to ensure that only valid user and book IDs are referenced.
- UNIQUE constraint on the `email` column of the `users` table to prevent duplicate email addresses.
- CHECK constraint on the `rating` column of the `reviews` table to ensure that ratings are between 1 and 5.
- INDEX on the `user_id` and `book_id` columns of the `reviews` table to improve query performance.

Step 5: Consider additional tables and relationships

Depending on the requirements of the review app, additional tables and relationships may be needed, such as:

- A `tags` table to store tags associated with books or reviews.
- A `categories` table to store categories associated with books or reviews.
- A `comments` table to store comments on reviews.
- A `votes` table to store votes on reviews.

By following these steps, a robust and well-designed schema for the review app can be created.

Comment: While Llama typically tends to be the more thorough model with most coding tasks, it seems that Claude's understanding of SQL is superior to Llama, allowing it to consider more important criteria.

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

```java
public int getLength(String s) {
    return s.length(); // What if s is null?
}
```

Selected Models: ChatGPT and Claude

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Identify any null dereference risk: (code)

ChatGPT Output:

Yes, there **is a null dereference risk** in the provided method. Here's why:

Analysis

Your code:

```java
public int getLength(String s) {
    return s.length(); // What if s is null?
}
```

If `s` is `null` when this method is called, the code will attempt to call the method `.length()` on a null reference. This will result in a **NullPointerException** being thrown at runtime.

Example Scenario

```java
String s = null;
int len = getLength(s); // This will throw a NullPointerException!
```

How to Fix

You can **handle the null case explicitly**, for example by returning `0` (or some other appropriate value), or you can throw an `IllegalArgumentException`:

Option 1: Return 0 if null

```java
public int getLength(String s) {
    if (s == null) {
        return 0;
    }
    return s.length();
}
```

Option 2: Throw a custom exception

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

Claude Output:

In this code, there's a significant null dereference risk:

```java
public int getLength(String s) {
    return s.length(); // What if s is null?
}
```

If `s` is `null`, calling `s.length()` will throw a `NullPointerException` because you're trying to invoke a method on a null reference.

To fix this issue, you should add a null check before attempting to access the `length()` method:

```java
public int getLength(String s) {
    if (s == null) {
        return 0; // or another appropriate default value
        // alternatively: throw new IllegalArgumentException("Input string cannot be n
    }
    return s.length();
}
```

This modified version either:

1. Returns a default value (like 0) when the input is null, or
2. Throws an explicit exception with a helpful message

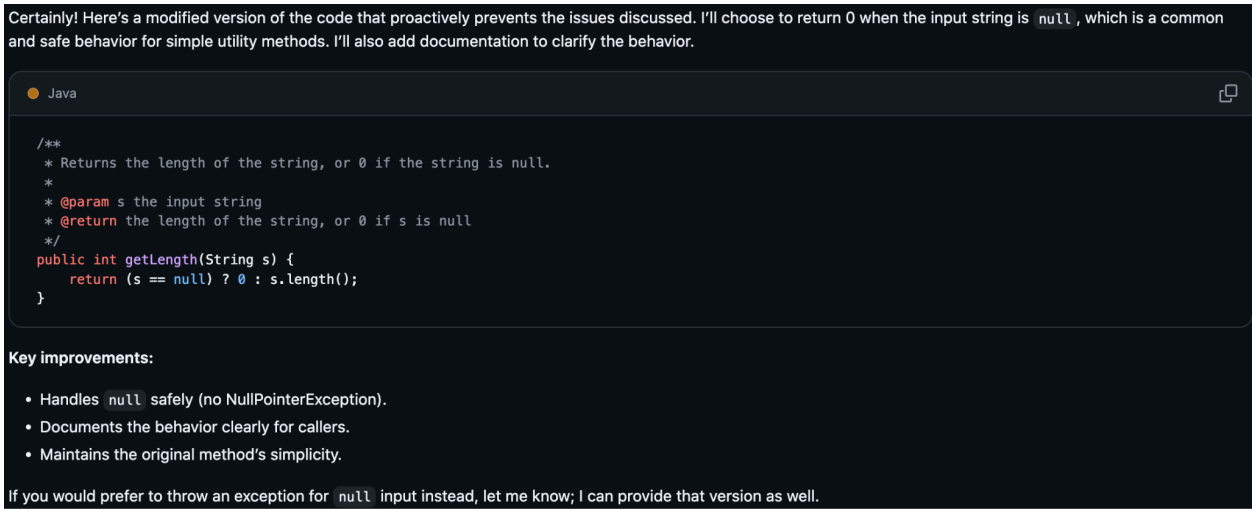
Which approach you choose depends on your application's requirements - whether a null input should be treated as an error condition or handled gracefully with a default return value.

Comment: Both correctly identify that there would be a null pointer exception and recommend the same solution. Either return 0 or a new custom exception.

Prompting Strategy 2: Prompt Chaining

Prompt 1: You are a code reviewer looking over one of your subordinates' work.
Prompt 2: Examine the following code and comment on what possible issues could arise: (code)
Prompt 3: Proactively modify the code to prevent these possible issues.

ChatGPT Output:



Claude Output: /**
* Returns the length of the provided string.
*
* @param s the string to measure
* @return the length of string s
* @throws NullPointerException if s is null
*/
public int getLength(String s) {
Objects.requireNonNull(s, "Input string cannot be null");
return s.length();
}

Comment: Both Claude and ChatGPT offer the same two solutions even with the prompting, however analysis is a little more in-depth upon the second step. Claude immediately provides both solutions, while ChatGPT defaults to the return 0 solution.

8. **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
```

Selected Models: Gemini and LLaMA

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Improve the parser to support quoted fields: (Code)
Gemini Output: import csv
import io

def parse_csv_line(line):
"""
Parses a CSV line, correctly handling quoted fields.

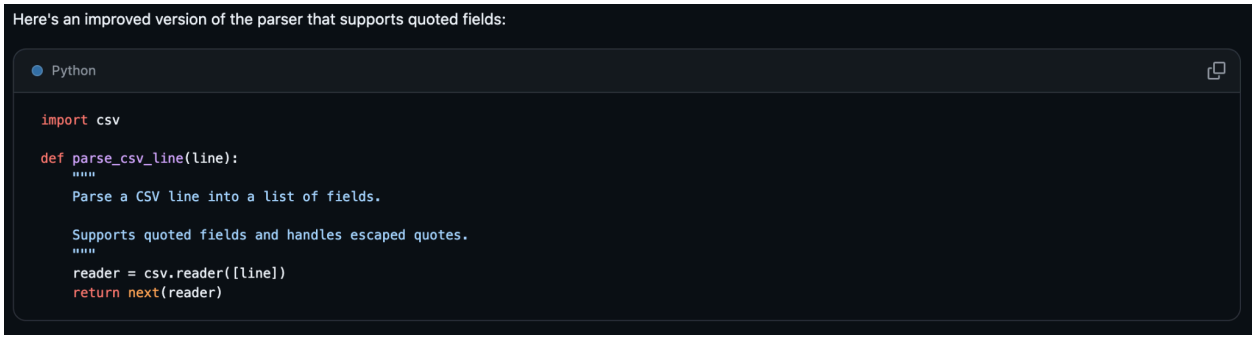
```
    """
    f = io.StringIO(line)
    reader = csv.reader(f)
    for row in reader:
        return row
    return []

# Example usage:
line1 = "field1,field2,field3"
result1 = parse_csv_line(line1)
print(f"Parsing '{line1}': {result1}")

line2 = 'field1,"quoted, field",field3'
result2 = parse_csv_line(line2)
print(f"Parsing '{line2}': {result2}")

line3 = '"another quoted, field",field2,"yet another, quoted field"'
result3 = parse_csv_line(line3)
print(f"Parsing '{line3}': {result3}")

line4 = 'field1,"quoted field with ""double quotes""",field3'
result4 = parse_csv_line(line4)
print(f"Parsing '{line4}': {result4}")
```

Instead of simply splitting the line by commas, this improved `parse_csv_line` function leverages Python's built-in `csv` module. Here's how it works:

1. `import csv` **and** `import io`**:** We import the necessary modules. The `csv` module provides functionality for working with CSV files, and `io` allows us to treat a string as a file-like object.
2. `io.StringIO(line)`**:** We create an in-memory text buffer (`StringIO`) from the input `line`. This allows the `csv.reader` to process the string as if it were reading from a file.
3. `csv.reader(f)`**:** We create a `csv.reader` object, passing it the file-like object `f`. The `csv.reader` is designed to handle the complexities of CSV parsing, including quoted fields and different delimiters (though we're assuming the default comma delimiter here).
4. `for row in reader:`**:** The `csv.reader` iterates over the rows in the CSV data. Since our input is a single line, this loop will typically execute only once.
5. `return row`**:** For each row (which is a list of fields), we return it.
6. `return []`**:** If the input line is empty, the loop won't execute, and we return an empty list.

This approach correctly handles commas within quoted fields and also deals with escaped double quotes (though the example doesn't explicitly show it, the `csv` module is capable of handling them).

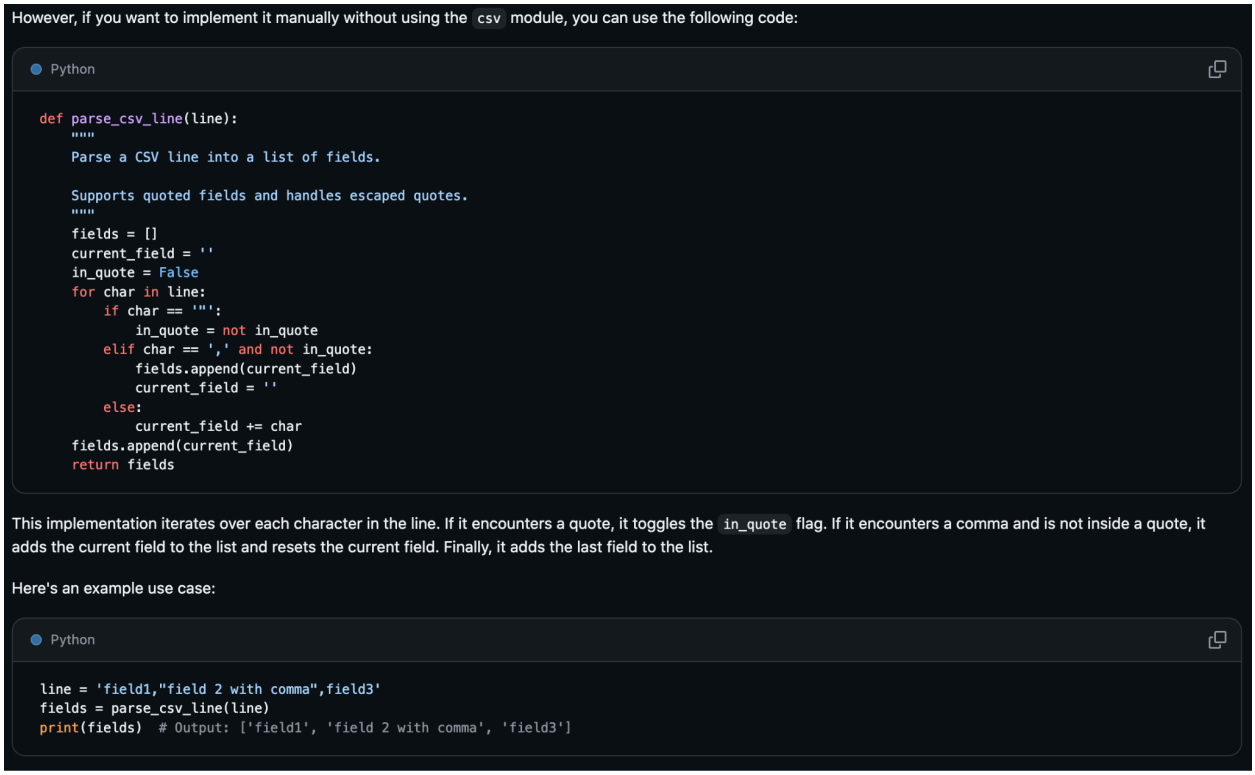
LLaMA Output:



Here's an improved version of the parser that supports quoted fields:

Python

```python
import csv

def parse_csv_line(line):
    """
    Parse a CSV line into a list of fields.

    Supports quoted fields and handles escaped quotes.
    """
    reader = csv.reader([line])
    return next(reader)
```



However, if you want to implement it manually without using the `csv` module, you can use the following code:

Python

```python
def parse_csv_line(line):
    """
    Parse a CSV line into a list of fields.

    Supports quoted fields and handles escaped quotes.
    """
    fields = []
    current_field = ''
    in_quote = False
    for char in line:
        if char == '"':
            in_quote = not in_quote
        elif char == ',' and not in_quote:
            fields.append(current_field)
            current_field = ''
        else:
            current_field += char
    fields.append(current_field)
    return fields
```

This implementation iterates over each character in the line. If it encounters a quote, it toggles the `in_quote` flag. If it encounters a comma and is not inside a quote, it adds the current field to the list and resets the current field. Finally, it adds the last field to the list.

Here's an example use case:

Python

```python
line = 'field1,"field 2 with comma",field3'
fields = parse_csv_line(line)
print(fields)  # Output: ['field1', 'field 2 with comma', 'field3']
```

Comment: The different Models actually took a much different approach to solve this problem. Llama iterated through each character individually and detected whether it could be inside a quote or a quotation mark itself, while Gemini uses a library which can handle the parsing for us.

Prompting Strategy 2:

Prompt:

CSV Parser Improvement Task

You need to improve a basic CSV parser function to properly handle quoted fields. Here's the current implementation that needs to be fixed:

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

Problem:

This simple implementation doesn't correctly handle quoted fields in CSV data. Quoted fields may contain commas that should not be treated as field separators.

Examples:

Example 1: Basic parsing without quotes

Input:

```
apple,banana,cherry
```

Current output (correct):

```
['apple', 'banana', 'cherry']
```

Example 2: Parsing with quoted fields

Input:

```
apple,"banana,with,commas",cherry
```

Current output (incorrect):

```
['apple', '"banana', 'with', 'commas"', 'cherry']
```

Expected output:

```
['apple', 'banana,with,commas', 'cherry']
```

Example 3: Parsing with escaped quotes inside quoted fields

Input:

```
apple,"banana ""special"" fruit",cherry
```

Current output (incorrect):

['apple', '"banana ""special"" fruit"', 'cherry']

Expected output:

['apple', 'banana "special" fruit', 'cherry']

Your task:

Improve the `parse_csv_line` function to correctly handle:

1. Regular fields separated by commas
2. Quoted fields that may contain commas (commas inside quotes should not split fields)
3. Escaped quotes within quoted fields (two double quotes in a row inside a quoted field represent one double quote character)

Use Python's built-in libraries or implement your own parsing logic. Consider edge cases and provide a solution that's both correct and efficient.

Gemini Output:

```
import csv
import io

def parse_csv_line(line):
    """
    Parses a CSV line, correctly handling quoted fields and escaped quotes.
    """
    f = io.StringIO(line)
    reader = csv.reader(f)
    for row in reader:
        return row
    return []

# Example Usage:
line1 = "apple,banana,cherry"
result1 = parse_csv_line(line1)
print(f"Input: '{line1}'\nCurrent Output: {result1}\nExpected Output: ['apple', 'banana', 'cherry']\n")

line2 = 'apple,"banana,with,commas",cherry'
```

```
result2 = parse_csv_line(line2)
print(f"Input: '{line2}'\nCurrent Output: {result2}\nExpected Output: ['apple',
'banana,with,commas', 'cherry']\n")

line3 = 'apple,"banana ""special"" fruit",cherry'
result3 = parse_csv_line(line3)
print(f"Input: '{line3}'\nCurrent Output: {result3}\nExpected Output: ['apple', 'banana
\"special\" fruit', 'cherry']\n")

line4 = '"leading and trailing quotes",field2,"another ""quoted"" field"'
result4 = parse_csv_line(line4)
print(f"Input: '{line4}'\nCurrent Output: {result4}\nExpected Output: ['leading and trailing
quotes', 'field2', 'another \"quoted\" field']\n")

line5 = 'field1,"embedded\nnewline",field3'
result5 = parse_csv_line(line5)
print(f"Input: '{line5}'\nCurrent Output: {result5}\nExpected Output: ['field1',
'embedded\\nnewline', 'field3']\n")
```

Explanation of the Improvement:

1. `import csv` **and** `import io`**:**
   - The `csv` module provides the necessary tools for parsing CSV formatted data.
   - The `io` module (specifically `io.StringIO`) allows us to treat a string as a file-like object, which is what the `csv.reader` expects as input.
2. `f = io.StringIO(line)`**:**
   - We create an in-memory text buffer (`StringIO`) from the input `line`. This allows the `csv.reader` to process the string as if it were reading from a file.
3. `reader = csv.reader(f)`**:**
   - We create a `csv.reader` object, passing it the file-like object `f`. The `csv.reader` is specifically designed to handle CSV parsing rules, including:
     - **Quoted Fields:** It recognizes fields enclosed in double quotes (`"`) and treats any commas within those quotes as part of the field value, not as separators.
     - **Escaped Quotes:** It correctly interprets two consecutive double quotes (`""`) within a quoted field as a single literal double quote (`"`).
4. `for row in reader:`**:**
   - The `csv.reader` iterates over the rows in the CSV data. Since we are processing a single line, this loop will typically execute only once, yielding a list of the parsed fields.
5. `return row`**:**
   - We return the parsed `row`, which is a list of strings representing the individual fields.
6. `return []`**:**
   - If the input `line` is empty, the loop won't execute, and we return an empty list.

Why this approach is better:

- **Correctness:** It adheres to the standard CSV parsing rules, correctly handling quoted fields and escaped quotes, which the simple `split(',')` cannot do.
- **Efficiency:** The `csv` module is implemented in C and is highly optimized for CSV processing. It's generally more efficient and less error-prone than trying to implement a robust CSV parser from scratch using regular expressions or manual string manipulation.
- **Readability and Maintainability:** Using the built-in `csv` module makes the code cleaner, more readable, and easier to maintain, as it clearly expresses the intent of parsing CSV data.

This improved `parse_csv_line` function effectively addresses the requirements of handling quoted fields and escaped quotes in CSV data.

LLaMA Output:

```
def parse_csv_line(line):
    """
```

```
Parse a CSV line into a list of fields.

Supports quoted fields, commas inside quotes, and escaped quotes.
"""
fields = []
current_field = ''
in_quote = False
for char in line:
    if char == '"':
        in_quote = not in_quote
        if in_quote:
            current_field += char
        else:
            # Remove the trailing quote
            if current_field and current_field[-1] == '"':
                current_field = current_field[:-1]
            fields.append(current_field)
            current_field = ''
    elif char == ',' and not in_quote:
        fields.append(current_field)
        current_field = ''
    else:
        current_field += char
# Handle the last field
if current_field:
    if in_quote:
        # Remove the trailing quote
        if current_field and current_field[-1] == '"':
            current_field = current_field[:-1]
        fields.append(current_field)
    else:
        fields.append(current_field)
return fields
```

Comment: Llama seems to refuse to use pre-made libraries, while Gemini uses them to keep the answer short and sweet. With this few shot prompting, both models explained their answers more. Perhaps a longer user input typically leads to a longer output from the AI assistant.

9. **Data Class to API Conversion (Kotlin)** Convert the data class to a REST API using Ktor:

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

// TODO: Create GET and POST endpoints using Ktor
```

Selected Models: LLaMA and ChatGPT

Prompting Strategy 1: Zero-Shot Prompting
Prompt:
ChatGPT Output:

```
import io.ktor.server.engine.*
import io.ktor.server.netty.*
import io.ktor.server.application.*
import io.ktor.server.request.*
import io.ktor.server.response.*
import io.ktor.server.routing.*
import io.ktor.server.plugins.contentnegotiation.*
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import io.ktor.serialization.kotlinx.json.*

@Serializable
data class Product(
    val id: Int,
    val name: String,
```

```
    val price: Double
)

fun main() {
    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            json(Json { prettyPrint = true })
        }

        // In-memory storage (thread-unsafe, for example only!)
        val products = mutableListOf<Product>()

        routing {
            // GET /products -> returns all products
            get("/products") {
                call.respond(products)
            }

            // POST /products -> creates a new product
            post("/products") {
                val product = call.receive<Product>()
                products.add(product)
                call.respond(mapOf("status" to "Product added", "product" to product))
            }
        }
    }.start(wait = true)
}
```

LLaMA Output:

```
import io.ktor.application.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.http.*
import io.ktor.gson.*
import io.ktor.serialization.*

fun Application.module() {
    install(io.ktor.serialization.jackson)
    install(io.ktor.gson.GsonConverter)
    install(io.ktor.features.CORS)

    val products = mutableListOf(
        Product(1, "Laptop", 999.99),
        Product(2, "Phone", 499.99),
```

```
        Product(3, "Tablet", 699.99)
    )

    routing {

        // GET endpoint to retrieve all products
        get("/products") {
            call.respond(products)
        }

        // GET endpoint to retrieve a product by id
        get("/products/{id}") {
            val id = call.parameters["id"]?.toInt()
            val product = products.find { it.id == id }
            if (product != null) {
                call.respond(product)
            } else {
                call.response.status(HttpStatusCode.NotFound)
            }
        }

        // POST endpoint to create a new product
        post("/products") {
            val product = call.receive<Product>()
            products.add(product)
            call.respond(HttpStatusCode.Created)
        }
    }
}

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

fun main() {
    val ktor pduNet = embeddedServer(Application::main)
    pduNet.start()
}
```

Comment: Much liberty is taken by each model with this prompt, as they use different main methods and make different assumptions about the desired applications of the REST API.

Prompting Strategy 2: Chain-Of-Thought
Prompt:

ChatGPT Output:

Step 1: Analyze the Problem

You have a Kotlin data class:

Kotlin

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

Goal:

- Build a REST API using Ktor that provides:
  - GET endpoint: List all products
  - POST endpoint: Add a new product

Step 2: Explore Implementation Possibilities

When designing a REST API in Ktor, there are several options and questions to consider:

A) Data Storage

1. **In-memory List**
   - Easiest for prototyping and demos.
   - Data is lost on server restart.
2. **File-based Storage**
   - Products are saved as JSON/YAML in files.
   - More persistent, but complex for concurrent access.
3. **Database Storage**
   - Use something like H2, SQLite, or Postgres.
   - Suitable for production apps, but requires database setup.

For simplicity, #1 (in-memory) is usually the best way to start for learning/prototyping.

B) Serialization

1. **Kotlinx.serialization**
   - Official, works well, but requires `@Serializable`.
2. **Jackson**
   - Widely used, especially if you have Java codebases.

Kotlinx.serialization is preferred for idiomatic Kotlin projects.

C) Id Management

1. **Client-supplied IDs**
   - Trust the user to supply unique IDs.
   - Easy, but error-prone.
2. **Server-generated IDs**
   - Server increments IDs automatically.
   - Safer, usually the best practice.

For safety, you often want the server to handle IDs.

D) Ktor Routing and Structure

1. **Inline Routing**
   - All logic inside `routing { ... }` block.
   - Fine for small projects.
2. **Modular Structure**
   - Extract routes and business logic to separate files/classes.
   - Better for scaling the codebase.

For this example, I'll show inline for clarity.

E) Validation

- Should we validate product data (e.g. positive price)?
- Returning error messages for invalid input is good practice.

Step 3: Choose an Approach

For a straightforward demo:

- In-memory list for storage.
- Server-generated IDs for new products.
- Serialization with kotlinx.serialization.
- Inline routing.
- Minimal (but present) validation.

Step 4: Implementation

Here's the solution with explanations in comments:

```
import io.ktor.server.engine.*
import io.ktor.server.netty.*
import io.ktor.server.application.*
```

```
import io.ktor.server.request.*
import io.ktor.server.response.*
import io.ktor.server.routing.*
import io.ktor.server.plugins.contentnegotiation.*
import io.ktor.serialization.kotlinx.json.*
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json

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

// Data class for incoming POST (without ID, since server generates it)
@Serializable
data class NewProduct(
    val name: String,
    val price: Double
)

fun main() {
    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            json(Json { prettyPrint = true })
        }

        val products = mutableListOf<Product>()
        var idCounter = 1

        routing {
            get("/products") {
                call.respond(products)
            }

            post("/products") {
                val newProduct = call.receive<NewProduct>()

                //undefined
```

LLaMA Output:

Let's break down the problem statement and explore all possible possibilities.

1. **Data Validation**:
    - We should validate the data received in both GET and POST requests to ensure it conforms to the expected data class. This helps prevent errors and ensures data consistency.
    - One approach is to use Ktor's built-in `validate` function, but it's available for HTTP clients, not directly for server-side validation.
    - Another option is to use Kotlin's validation functions from the Kotlin standard library, such as `validate` and `required` functions from the `kotlinx.serialization` library.
2. **Error Handling**:
    - We should handle potential errors that may occur when processing requests, such as database errors or JSON deserialization errors.
    - We can leverage Ktor's built-in error handling mechanisms, like the `Constraints` plugin, to handle various types of errors and exceptions.
    - Another option is to use a custom error handler that catches exceptions and returns a standard error response.
3. **Database Integration**:
    - As per the problem statement, we have the data class `Product` with an `id`, `name`, and `price`. We could store these products in a database, like Postgres or MongoDB, to persist data between different server runs.
    - We would need to integrate the Ktor application with the chosen database using a database driver, like `kotlinxpostgres` or `kotlinxmongo`. This would help us to perform CRUD operations on the database.

4. **Product Existence Check:**
    - In the GET endpoint that returns products by their id, we should check if the product exists in the database before returning it to the client.
    - Similarly, when creating a new product using the POST endpoint, we should check if a product with the same id already exists in the database and return an error if so.
5. **Use a specialised Data Access Object (DAO)**:
    - As the complexity of your application increases, it will be better to encapsulate database operations in a DAO which holds the query as string equivalents.
    - The main benefit of doing this is that it will keep the code organized if you were to switch from a SQL database to an ORM database (Object Relational Mapping).
6. **REST API Restrictions**:
    - Typically, we wouldn't allow anyone to POST any data and expect the server to handle it. Major limitations are usually considered:
        - Consider the potential damage that incorrect information can cause. Thus, don't allow unauthenticated actions which write to the server.
        - One technique is to delay the response by delta seconds.
7. **General errors on JSON requirement**:
    - Consider adding error parameters for fields that can be null.
    - Then, pass strings which generally describe the validation result for the fields

With all these considerations, the implementation of the data class to a REST API using Ktor can look something like this:

```
import io.ktor.application.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.http.*
import io.ktor.gson.*
import io.ktor.serialization.*

fun main() {
    val ktorPort = System.getenv("KIOR_PORT")?.toInt() ?: 8080
    val databaseUrl = System.getenv("DATABASE_URL") ?:
"jdbc:h2:mem:mydb;DB_CLOSE_DELAY=-1"

    embeddedServer(Netty, port = ktorPort) {
        install(ContentNegotiation) {
            gson {
                setPrettyPrinting()
                registerTypeAdapter(Product::class, ProductListAdapter())
            }
        }
```

```
        val dao = ProductDAO(databaseUrl)

        routing {
            val products = dao.getProducts()

            get("/products") {
                call.respond(convert(products))
            }

            get("/products/{id}") {
                val product = dao.getProduct(call.parameters["id"].toString())
                if (product != null) {
                    call.respond(convert(product))
                } else {
                    val stringResponse = "Product not found"
                    call.respond(HttpStatusCode.NotFound, stringResponse)
                }
            }

            post("/products") {
                val newItem = call.receive<Product>()
                try {
                    dao.createProduct(newItem)
                    call.respond(HttpStatusCode.Created)
                } catch (e: Exception) {
                    val stringResponse = e.message
                    call.respond(HttpStatusCode.StatusBadRequest, stringResponse) }
            }
        }
    }.start()
}

data class Product(
    val id: Int = 0,
    val name: String = "",
    val price: Double = 0.0
)

data class ProductRequest(
    val name: String = "",
    val price: Double = 0.0
)
```

```
object ProductListAdapter {
    @JvmStatic
    @Suppress("UNCHECKED_CAST")
    @JsonDeserialize(`as` = `Deserializer`.
            <`Product`>)
    @JsonSerialize(`type` = `Serializer`.
            <`Product`>)
    fun comicDeserializeData() {
        // Deserialization magic
    }
}

class ProductDAO(private val url: String) {
    init {
        println("Connecting to $url")
    }

    fun createNewTable() {
        // execute SQL for creating table here
    }

    fun getProducts(): List<Product> {
        // execute SQL to fetch all products here
    }

    fun getProduct(name: String): Product? {
        // execute SQL to fetch the product by name here
    }

    fun createProduct(product: Product) {
        // execute SQL to insert product into the database here
        createNewTable()
        println(product)
    }
}
```

Comment: I am not completely familiar with REST API, but both models were able to come up with unique considerations that the other one did not. Both considered certain common issues such as validation at different points or in different ways.

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

```python
def reverse_words(sentence):
    return ' '.join(sentence.split()[::-1])
```

Selected Models: Claude and Gemini

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Write a brief summary of the function: (function)
Claude Output:

This function `reverse_words` takes a string parameter `sentence` and returns a new string with the words in the sentence reversed in order, while maintaining the original order of characters within each word.

It works by:

1. Splitting the input sentence into a list of words using `split()`
2. Reversing the order of the words in the list with the slice notation `[::-1]`
3. Joining the reversed list back into a string with spaces between words using `' '.join()`

For example, if given "Hello world example", it would return "example world Hello".

Gemini Output:

The `reverse_words` function takes a sentence as input, splits it into individual words, reverses the order of these words, and then joins them back together with spaces to form a new sentence with the words in reversed order.

Comment: Both provide very straightforward breakdowns of the function, but claude dives slightly more into the technical behavior of the function.

Prompting Strategy 2: Chain-of-Thought Prompting
Prompt: Write a brief summary of the function, but take me through each line individually (function)
Claude Output:

Function Summary: `reverse_words`

This function reverses the order of words in a sentence while preserving the order of characters within each word.

Line-by-Line Breakdown:

```python
def reverse_words(sentence):
```

- Defines a function named `reverse_words` that takes one parameter, `sentence`, which is expected to be a string.

```python
    return ' '.join(sentence.split()[::-1])
```

This single line does all the work through a sequence of operations:

1. `sentence.split()` - Splits the input string into a list of words using whitespace as the delimiter. For example, "Hello world" becomes `["Hello", "world"]`.
2. `[::-1]` - This is Python's slice notation for reversing a sequence. It takes the list from the previous step and reverses the order of the words. For example, `["Hello", "world"]` becomes `["world", "Hello"]`.
3. `' '.join(...)` - Takes the reversed list of words and joins them back together into a single string, with spaces between each word. For example, `["world", "Hello"]` becomes "world Hello".

The function returns this final string with the word order reversed.

Gemini Output:

Okay, let's break down the `reverse_words` function line by line:

Python

```
def reverse_words(sentence):
```

- `def reverse_words(sentence):` : This line defines a function named `reverse_words`. It takes one argument, which is expected to be a string representing a sentence, and this argument is named `sentence`.

Python

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

- `sentence.split()` : This part of the line calls the `split()` method on the input `sentence` string. The `split()` method, when called without any arguments, splits the string at whitespace characters (spaces, tabs, newlines) and returns a list of the individual words. For example, if `sentence` is `"hello world"`, `sentence.split()` would return `['hello', 'world']`.
- `[::-1]` : This is a Python slice notation. When applied to a list, `[::-1]` creates a reversed copy of that list. So, if the result of `sentence.split()` was `['hello', 'world']`, then `sentence.split()[::-1]` would be `['world', 'hello']`.
- `return ...` : Finally, the `return` statement sends the resulting reversed sentence back as the output of the `reverse_words` function.

In essence, the function cleverly uses string manipulation and list slicing to achieve word reversal in a concise single line of code. It first breaks the sentence into words, reverses the order of those words in a list, and then reconstructs the sentence with the reversed word order.

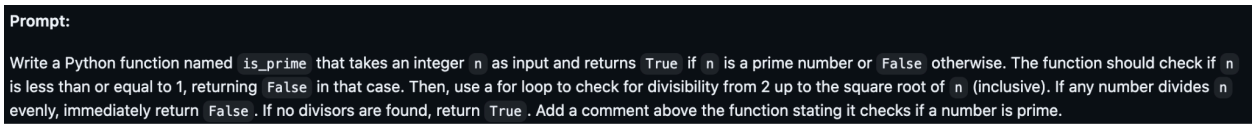
Comment: Both models go even further than each line, moving token by token. I haven't had something broken down like this to me since my first CS class in high school.

11. **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
```

Selected Models: ChatGPT and Gemini

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Write a prompt that could generate the code: (prime function)
ChatGPT Output:



Gemini Output:

Here are a few prompt options that could generate the code you provided:

Prompt 1 (Specific):

> **Write a Python function called `is_prime` that takes an integer `n` as input and returns `True` if `n` is a prime number, and `False` otherwise. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. Optimize the function by only checking for divisors up to the square root of `n`.**

Prompt 2 (Less Specific):

> **Write a Python function to determine if a given integer is a prime number. The function should return a Boolean value.**

Prompt 3 (With Constraints):

> **Write a Python function that efficiently checks if a given positive integer `n` is a prime number. The function should be named `is_prime` and take one integer argument. It should return `True` if the number is prime, and `False` otherwise. Ensure that the function handles the case where n is less than or equal to 1 correctly.**

Prompt 4 (Code Completion Style):

> **Complete the following Python function:**

```
def is_prime(n):
    """Checks if a number is prime."""
    if n <= 1:
        return False
    #  ... complete the rest of the function ...
```

Comment: ChatGPT provides a much more concise answer while Gemini unnecessarily creates a bunch of options for a very simple task.

Prompting Strategy 2:
Prompt:
ChatGPT Output 1:

Sure! Here is a prompt you could use to generate the code you provided:

Prompt:

Write a Python function called `is_prime(n)` that returns True if the input number `n` is a prime number, and False otherwise. The function should:

- Return False if `n` is less than or equal to 1.
- Use a for loop to check divisibility from 2 up to the square root of `n`.
- Return False as soon as a divisor is found; otherwise, return True at the end.
  Add a comment above your function stating its purpose.

ChatGPT Output 2:

Prompt:
Write a Python function called `is_prime` that takes an integer `n` and returns `True` if `n` is a prime number and `False` otherwise. The function should return `False` for numbers less than or equal to 1, and for other numbers, check for divisibility up to the square root of `n`.

ChatGPT Output 3:

Write a Python function called `is_prime` that checks if a given number `n` is a prime number. If `n` is less than or equal to 1, it should return `False`. Otherwise, it should return `True` only if `n` is divisible by no integers other than 1 and itself. Use a for loop that checks divisibility for all numbers from 2 up to the square root of `n` (inclusive).

Gemini Output 1:

Write a Python function called `is_prime` that takes one integer argument `n`. The function should return `True` if the input integer is a prime number and `False` otherwise.

The primality check should include the following optimizations:

- Numbers less than or equal to 1 are not prime.
- The function should iterate only up to the square root of `n` to check for divisors.

Gemini Output 2:

Write a Python function called `is_prime` that takes an integer `n` as input and returns `True` if `n` is a prime number, and `False` otherwise. Ensure the function efficiently checks for primality by handling the base case where `n` is less than or equal to 1 and by only checking for divisors up to the square root of `n`.

Gemini Output 3:

Generate a Python function named `is_prime` that accepts a single integer argument, `n`. This function should determine if the given integer is a prime number and return a boolean value (`True` if prime, `False` otherwise). The implementation should include a check for numbers less than or equal to 1 (which are not prime) and optimize the divisibility check by only iterating up to the square root of `n`.

Comment: Both models provide slightly different prompts with the same general guidance, adding or removing clarification from each output. ChatGPT 4.1 now seems to be the most concise of the models I've looked at so far.

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

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

Selected Models: Claude and LLaMA

Prompting Strategy 1: Zero-Shot Prompting
Prompt:
Claude Output:

The bug was in the `range(1, n)` which:

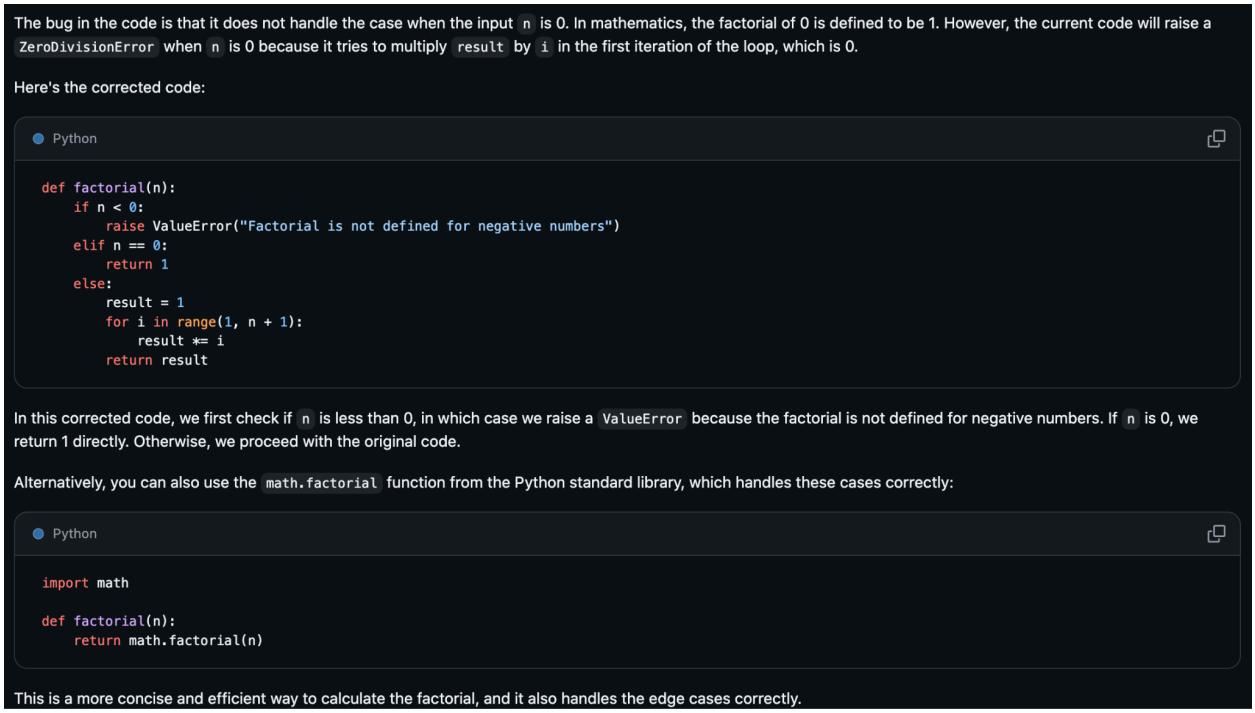
1. Creates a sequence of numbers from 1 to n-1, not including n
2. When n is 0, range(1, 0) produces an empty sequence, so the loop doesn't execute
3. The function returns 1, which is technically correct for 0! = 1, but not because of the right calculation

The fix changes the range to `range(1, n + 1)` so that:

1. It creates a sequence from 1 to n, inclusive
2. When n is 0, range(1, 1) still produces an empty sequence, so the function correctly returns 1
3. For all other positive integers, it properly calculates the factorial by multiplying from 1 to n

This maintains the mathematical definition of factorial where 0! = 1 and n! = n × (n-1) × ... × 2 × 1

LLaMA Output:



Comment: Claude changes the range to be n+1, and Llama provides the same solution. Llama this time however suggests a python library, which in one of the previous comparisons it staunchly refused to do.

Prompting Strategy 2: Few-Shot Prompting

Prompt: Prompt:

You are a helpful AI assistant that can correct code. Here are some examples of buggy code and their corrected versions:

Example 1:

- **Buggy Code:**

```
def sum_numbers(numbers):
    total = 0
    for number in numbers:
        total += number
    return  # Bug: No return value if the list is empty
```

-
- **Corrected Code:**

```
def sum_numbers(numbers):
    total = 0
    for number in numbers:
```

```
        total += number
    return total
```

-

Example 2:

- **Buggy Code:**

```
def divide(x, y):
    if y == 0:
        print("Cannot divide by zero") # Bug: Should raise an exception
    else:
        return x / y
```

-
- **Corrected Code:**

```
def divide(x, y):
    if y == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    else:
        return x / y
```

-

Example 3:

- **Buggy Code:**

```
def greet(name):
    print("Hello, " + Name + "!") # Bug: Variable Name is not defined
```

-
- **Corrected Code:**

```
def greet(name):
    print("Hello, " + name + "!")
```

-

Now, correct the following code. Identify and fix the bug:

Buggy Code:

```
def factorial(n):
```

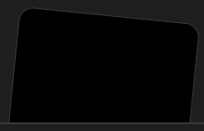
```
    result = 1
    for i in range(1, n):
        result *= i
    return result
```

Claude Output:

Looking at the provided code, I can identify the bug in the factorial function:

Corrected Factorial Function

Code



Bug Identified: The function uses `range(1, n)` which creates a sequence from 1 to n-1, missing the last number n itself in the multiplication.

Fix: Changed to `range(1, n + 1)` to include n in the range.

The corrected code properly calculates the factorial by:

1. Starting with result = 1
2. Multiplying by all integers from 1 up to and including n
3. For n = 0, the loop doesn't execute (range(1, 1) is empty), so it correctly returns 1, which is the mathematical definition of 0!

LLaMA Output:

The bug in the code is that it does not handle the case when the input `n` is 0. In mathematics, the factorial of 0 is defined to be 1. However, the current code will not return a value when `n` is 0 because the loop will not be executed.

Here's the corrected code:

Python

```python
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
```

However, a more concise and Pythonic way to calculate the factorial is to use a recursive function or the `math.factorial` function from the Python standard library:

Python

```python
import math

def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    else:
        return math.factorial(n)
```

Comment: Neither response in this situation is improved by the few-shot prompting, I believe this specific problem is too simple for it to show tangible effects.

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

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

void deleteNode(struct Node** head, int key) {
    // TODO: Implement node deletion
}
```

Selected Models: ChatGPT and Claude

Prompting Strategy 1: Zero-Shot Prompting
Prompt:
ChatGPT Output:

Here is how you can implement the `deleteNode` function to delete the first occurrence of a node with value `key` in a singly linked list in C:

C

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

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

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

    // If the head node itself holds the key
    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 previous node
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

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

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

    free(temp); // Free memory
}
```

Claude Output:

```
/**
 * Deletes a node with the specified value from a singly linked list
 *
 * @param head Double pointer to the head of the linked list
 * @param key The value to search for and delete
 */
void deleteNode(struct Node** head, int key) {
```

```
    // Check if list is empty
    if (*head == NULL) {
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

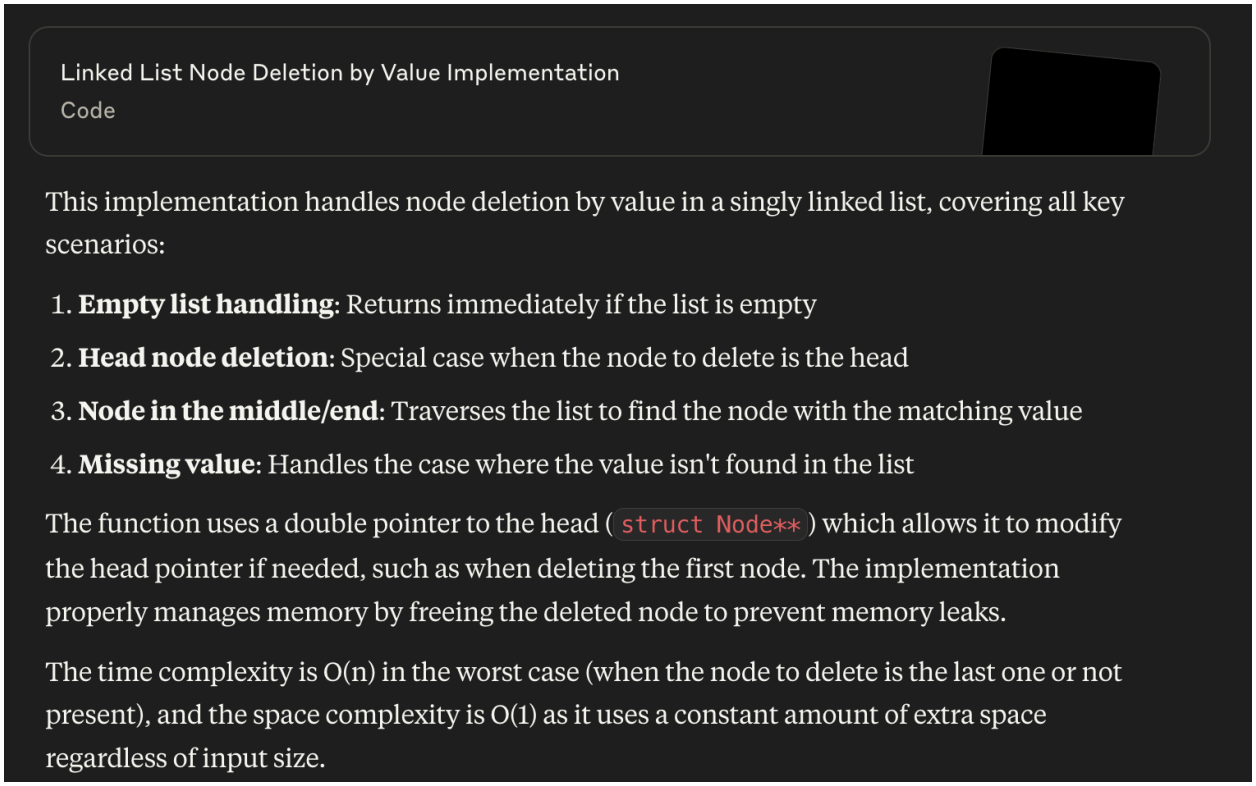
    // Case 1: If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key) {
        *head = temp->next;  // Change head
        free(temp);          // Free old head
        return;
    }

    // Case 2: Search for the key to be deleted, keep track of previous node
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

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

    // Case 3: Unlink the node from linked list
    prev->next = temp->next;

    // Free memory
    free(temp);
}
```



Comment: Claude provides a more thorough explanation of the design choices made within the function. It also provides more comments within the code it provides.

Prompting Strategy 2: Prompt Chaining
Prompt: Prompt Chain:

Prompt 1: Explain the Problem

> "You are a helpful AI assistant that can generate C code. First, explain the concept of a singly linked list and how to delete a node with a specific value. Provide an example linked list and illustrate the deletion process step by step. Use clear and concise language, avoiding jargon as much as possible. Assume the user is familiar with basic programming concepts but may not know about linked lists."

Prompt 2: Define the Function Signature

> "Now, provide the C function signature for a function called `deleteNode` that deletes a node with a specific value from a singly linked list. The function should take two arguments: a pointer to the head of the list (which may need to be updated if the first node is deleted) and the integer value of the node to be deleted. The function should not return any value (i.e., `void`).

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

void deleteNode(struct Node **head, int key) {
// TODO: Implement node deletion
}
```"

Prompt 3: Step-by-Step Implementation

"Provide a step-by-step implementation of the `deleteNode` function in C. For each step, explain the logic and provide the corresponding C code. Handle the following cases:

1. The list is empty.
2. The node to be deleted is the first node in the list.
3. The node to be deleted is in the middle or at the end of the list.
4. The node with the given value is not found in the list.

Ensure that the code updates the `next` pointers of the remaining nodes correctly and frees the memory of the deleted node to prevent memory leaks. The code should be well-commented and easy to understand."

Prompt 4: Complete Code Generation

"Now, combine the function signature and the step-by-step implementation to generate the complete C code for the `deleteNode` function. Include all necessary comments and ensure that the code is properly formatted. Do not include any additional explanations or examples, just the final C code."

ChatGPT Output:

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

// Definition of the Node structure
struct Node {
    int data;
```

```
    struct Node *next;
};

// Function to delete the first node with the given key
void deleteNode(struct Node **head, int key) {
    // Step 1: Check if the list is empty
    if (*head == NULL) {
        return;
    }

    struct Node *temp = *head;
    struct Node *prev = NULL;

    // Step 2: If the head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Move the head pointer to the next node
        free(temp);         // Free the old head
        return;
    }

    // Step 3: Search for the node to be deleted
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // Step 4: If the key was not found in the list
    if (temp == NULL) {
        return;
    }

    // Step 5: Unlink the node from the list
    prev->next = temp->next;

    // Step 6: Free the memory of the deleted node
    free(temp);
}
```

Claude Output:

```
/**
 * Deletes the first occurrence of a node with the specified value from a singly linked list.
 *
 * @param head Pointer to the pointer to the head of the list
```

```
 * @param key  The value to search for and delete
 */
void deleteNode(struct Node **head, int key) {
    // Case 1: Check if the list is empty
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node *temp = *head;  // Create a temporary pointer to the current node

    // Case 2: Check if the head node contains the key
    if (temp->data == key) {
        *head = temp->next;     // Update head to point to the second node
        free(temp);             // Free the old head node's memory
        printf("Node with value %d deleted.\n", key);
        return;
    }

    // Case 3 & 4: Find the node to delete in the middle or end of the list
    struct Node *current = *head;
    struct Node *prev = NULL;

    // Search for the node with the given key
    while (current != NULL && current->data != key) {
        prev = current;         // Keep track of the previous node
        current = current->next; // Move to the next node
    }

    // Case 4: If the key wasn't found in the list
    if (current == NULL) {
        printf("Node with value %d not found in the list.\n", key);
        return;
    }

    // Case 3: Node found in the middle or end of the list
    // Update the previous node's next pointer to skip over the node to delete
    prev->next = current->next;

    // Free the memory of the node to delete
    free(current);

    printf("Node with value %d deleted.\n", key);
}
```

Comment: I used gemini to create the chained prompts, but it resulted in both models being incredibly thorough with their thought process. The final solutions of both models are incredibly well thought out.

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

```python
def fibonacci(n):
    # TODO: Base cases and recursive call
    pass
```

Selected Models: Gemini and LLaMA

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Complete the recursive function for Fibonacci: (func)

Gemini Output:

Python

```
def fibonacci ( n ) :
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Here's how the code works:

1. **Base Cases:**
   - `if n <= 0:` : If `n` is 0 or negative, the Fibonacci number is defined as 0. This is the first base case that stops the recursion.
   - `elif n == 1:` : If `n` is 1, the Fibonacci number is defined as 1. This is the second base case.
2. **Recursive Step:**
   - `else: return fibonacci(n - 1) + fibonacci(n - 2)` : For any `n` greater than 1, the Fibonacci number is the sum of the Fibonacci numbers of the two preceding numbers (`n-1` and `n-2`). The function calls itself with these smaller values of `n`, breaking the problem down into smaller, self-similar subproblems until it reaches the base cases.

Important Note: While this recursive implementation is clear and directly reflects the mathematical definition of the Fibonacci sequence, it can be inefficient for larger values of `n` due to repeated calculations. For better performance with larger inputs, an iterative approach using a loop and storing previously calculated values is generally preferred.

LLaMA Output:

Here's the completed function:

Python

```python
def fibonacci(n):
    """
    Recursively calculates the nth Fibonacci number.

    :param n: The index of the Fibonacci number to calculate.
    :return: The nth Fibonacci number.
    """
    # Base cases
    if n == 0:
        return 0  # The 0th Fibonacci number is 0
    elif n == 1:
        return 1  # The 1st Fibonacci number is 1
    else:
        # Recursive call
        return fibonacci(n-1) + fibonacci(n-2)  # Calculate the nth Fibonacci number using the previous

# Example usage:
print(fibonacci(10))  # Output: 55
```

However, please note that this recursive implementation has a major issue. It has a time complexity of O(2^n), which is very inefficient for large values of n. This is because each Fibonacci calculation causes two additional calculations to be performed, leading to an exponential increase in the number of calculations.

A better approach would be to use memoization or dynamic programming to store and reuse previously calculated Fibonacci numbers. Here's an example of how you can modify the function to use memoization:

Python

```python
def fibonacci(n, memo = {}):
    """
    Recursively calculates the nth Fibonacci number using memoization.

    :param n: The index of the Fibonacci number to calculate.
    :param memo: A dictionary to store previously calculated Fibonacci numbers.
    :return: The nth Fibonacci number.
    """
    # Base cases
    if n in memo:
        return memo[n]  # Return the memoized value if it exists
    elif n == 0:
        return 0  # The 0th Fibonacci number is 0
    elif n == 1:
        return 1  # The 1st Fibonacci number is 1
    else:
        # Calculate the nth Fibonacci number and store it in the memo dictionary
        result = fibonacci(n-1, memo) + fibonacci(n-2, memo)
        memo[n] = result
        return result

# Example usage:
print(fibonacci(10))  # Output: 55
```

This implementation has a time complexity of O(n), which is much more efficient for large values of n.

Comment: Both models return a basic version of the recursive fibonacci sequence, but also note the time inefficiency with larger numbers. LLaMA however does suggest a dynamic programming approach and provides an implementation of it.

Prompting Strategy 2: Chain-of-Thought
Prompt: Consider the classic recursive fibonacci function coding problem. Explain step by step things to consider when creating a solution for this problem. Then, implement the most robust and efficient solution you can think of.

Gemini Output:

```python
def fibonacci_efficient_space_optimized(n):
    """
    Calculates the nth Fibonacci number using dynamic programming (bottom-up),
    optimized for space complexity.

    Args:
        n: A non-negative integer representing the desired Fibonacci number index.

    Returns:
        The nth Fibonacci number.
        Raises ValueError if n is negative.
    """
    if n < 0:
        raise ValueError("Input must be a non-negative integer")
    elif n <= 1:
        return n
    else:
        a, b = 0, 1
        for _ in range(2, n + 1):
            c = a + b
            a = b
            b = c
        return b

# Example usage
print(fibonacci_efficient_space_optimized(0))   # Output: 0
print(fibonacci_efficient_space_optimized(1))   # Output: 1
print(fibonacci_efficient_space_optimized(5))   # Output: 5
print(fibonacci_efficient_space_optimized(10))  # Output: 55
print(fibonacci_efficient_space_optimized(20))  # Output: 6765
```

```
# Example of handling invalid input
try:
    print(fibonacci_efficient_space_optimized(-1))
except ValueError as e:
    print(f"Error: {e}")
```

This space-optimized version maintains the $O(n)$ time complexity but reduces the space complexity to $O(1)$ because we only store a constant number of variables (`a` , `b` , and `c`). This is generally the most practical and efficient approach for calculating Fibonacci numbers for larger values of `n` .

Sources and related content

github.com

github.com

github.com

github.com

LLaMA Output:

```
def fibonacci(n):
    # Check if input is a non-negative integer
    if not isinstance(n, int) or n < 0:
        raise ValueError("Input must be a non-negative integer.")

    # Base cases
    if n == 0 or n == 1:
        return n

    # Matrix exponentiation method
    def multiply_matrices(A, B):
        result = [[0, 0], [0, 0]]
        for i in range(2):
            for j in range(2):
                for k in range(2):
                    result[i][j] += A[i][k] * B[k][j]
        return result

    def matrix_power(matrix, n):
        if n == 0:
            return [[1, 0], [0, 1]]
        if n == 1:
            return matrix
        if n % 2 == 0:
            half_pow = matrix_power(matrix, n // 2)
            return multiply_matrices(half_pow, half_pow)
        else:
            half_pow = matrix_power(matrix, n // 2)
            return multiply_matrices(multiply_matrices(half_pow, half_pow), matrix)

    # Calculate the matrix representing the Fibonacci sequence
    F = [[1, 1], [1, 0]]

    # Calculate the matrix raised to the power of n-1
    result_matrix = matrix_power(F, n - 1)

    # Return the (n-1)-th Fibonacci number
    return result_matrix[0][0]
```

Comment: When prompted explicitly to be robust, both models take a very advanced, but different approach to optimization. I'm not quite sure which one is more efficient.

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

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

Selected Models: LLaMA and ChatGPT

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Complete the class constructor: (class)
ChatGPT Output:

Certainly! Based on your prompt, the constructor should add `name`, `age`, and an **optional** `email`. Usually, you'd set parameters in the constructor and assign them to instance variables. Here's a way to do it:

Python

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

Explanation:

- `name` and `age` are required parameters.
- `email` is optional and defaults to `None` if not provided.

Example usage:

Python

```
p1 = Person("Alice", 30)
p2 = Person("Bob", 25, "bob@email.com")
```

Let me know if you need a version where the attributes are initialized without arguments!

LLaMA Output:

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

Let's break down what we've done:

- The constructor `__init__` takes in three parameters:
  - `name` : The person's name, which will be a string.
  - `age` : The person's age, which will be an integer.
  - `email` : The person's email address, which is optional and defaults to `None` . (We use `email=None` to indicate this.)
- Inside the constructor, we've added three instance variables:
  - `self.name` = `name`
  - `self.age` = `age`
  - `self.email` = `email`

Here's an example of how you might use this class:

Python

```python
p1 = Person("John Doe", 30)
print(p1.name)  # prints "John Doe"
print(p1.age)   # prints 30
print(p1.email) # prints None

p2 = Person("Jane Smith", 25, "jane.smith@example.com")
print(p2.name)       # prints "Jane Smith"
print(p2.age)        # prints 25
print(p2.email)      # prints "jane.smith@example.com"
```

Comment: Both create a constructor with an email that defaults to None, and both give examples of how to use the class even though I never prompted the model to do this.

Prompting Strategy 2: Prompt chaining
Prompt 1: Consider a class in python that represents a person, what might this class require?
Prompt 2: We want to be able to store this person's email address, but it is not required.
Prompt 3: Also make sure that name and age are recorded in the class.
Prompt 4: Create the class
ChatGPT Output:

```
class Person:
    def __init__(self, name, age, gender=None, address=None, email=None):
        self.name = name      # Required
        self.age = age        # Required
        self.gender = gender  # Optional
        self.address = address  # Optional
```

```
        self.email = email    # Optional

    def greet(self):
        print(f"Hello, my name is {self.name}.")

    def is_adult(self):
        return self.age >= 18

    def update_address(self, new_address):
        self.address = new_address
```

LLaMA Output:

```
class Person:
    def __init__(self, name, age, email=None, address=None, phone_number=None,
occupation=None):
        self.name = name
        self.age = age
        self.email = email
        self.address = address
        self.phone_number = phone_number
        self.occupation = occupation

    def get_info(self):
        info = f"{self.name} ({self.age})"
        if self.email:
            info += f", Email: {self.email}"
        if self.address:
            info += f", Address: {self.address}"
        if self.phone_number:
            info += f", Phone Number: {self.phone_number}"
        if self.occupation:
            info += f", Occupation: {self.occupation}"
        return info

    def update_info(self, name=None, age=None, email=None, address=None,
phone_number=None, occupation=None):
        if name:
            self.name = name
        if age:
            self.age = age
        if email:
            self.email = email
        if address:
            self.address = address
```

```
        if phone_number:
            self.phone_number = phone_number
        if occupation:
            self.occupation = occupation

    def __str__(self):
        return self.get_info()

# Example usage:
john = Person("John Doe", 30, "john.doe@example.com", "123 Main St",
"123-456-7890", "Software Engineer")
print(john.get_info())  # Output: John Doe (30), Email: john.doe@example.com, Address:
123 Main St, Phone Number: 123-456-7890, Occupation: Software Engineer

jane = Person("Jane Smith", 25, occupation="Marketing Manager")
print(jane.get_info())  # Output: Jane Smith (25), Occupation: Marketing Manager

john.update_info(email="johndoe2@example.com", address="456 Elm St")
print(john.get_info())  # Output: John Doe (30), Email: johndoe2@example.com,
Address: 456 Elm St, Phone Number: 123-456-7890, Occupation: Software Engineer
```

Comment: Both models when given looser outputs go above and beyond the

16. **Binary Search Completion (Java)** Complete the binary search implementation:

```java
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;
}
```

Selected Models: Claude and Gemini

Prompting Strategy 1: Zero-Shot Prompting
Prompt:
Claude Output:

```
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if (arr[mid] == target) {
```

```
            return mid;  // Found the target, return its index
        } else if (arr[mid] < target) {
            left = mid + 1;  // Target is in the right half
        } else {
            right = mid - 1;  // Target is in the left half
        }
    }

    return -1;  // Target not found in the array
}
```

Gemini Output:

```
public int binarySearch (int [] arr , int target ) {
    int left = 0 , right = arr . length - 1;
    while ( left <= right ) {
        int mid = left + (right - left) / 2; // Prevents potential overflow
        if (arr[mid] == target) {
            return mid; // Target found at index mid
        } else if (arr[mid] < target) {
            left = mid + 1; // Target is in the right half
        } else {
            right = mid - 1; // Target is in the left half
        }
    }
    return -1; // Target not found in the array
}
```

Comment: The solutions are identical other than Gemini accounting for an extreme edge case. In the event where the arrays smallest and largest numbers sum to a number which could overflow an int, it is able to handle the situation much better than Claude's solution.

Prompting Strategy 2: Three shot prompting

Prompt: Task: Complete the missing section in the Java binary search function by writing the comparison and adjustment logic. The function should return the index of `target` in `arr`, or `-1` if not found.

Example 1:

Java

```
public int findElement(int[] array, int value) {
    int start = 0, end = array.length - 1;
    while (start <= end) {
        int center = (start + end) / 2;
        // TODO: Complete this
    }
    return -1;
```

```
}
```

Completion:

Java

```
if (array[center] == value) {
    return center;
} else if (array[center] < value) {
    start = center + 1;
} else {
    end = center - 1;
}
```

Example 2:

Java

```
public int searchNumber(int[] nums, int key) {
    int low = 0, high = nums.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        // TODO: Implement bounds adjustment based on comparison
    }
    return -1;
}
```

Completion:

Java

```
if (nums[mid] == key) {
    return mid;
} else if (nums[mid] < key) {
    low = mid + 1;
} else {
    high = mid - 1;
}
```

Task:

Java

```
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;
}
```

Claude Output:

```
public int binarySearch (int [] arr , int target ) {
    int left = 0 , right = arr . length - 1;
    while ( left <= right ) {
```

```
        int mid = ( left + right ) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Gemini Output:

```
if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
```

Comment:
Both models correctly complete the function, and don't add any extra natural language, just returning the code. Claude does a better job of fitting the prompt as it only returns the code snippet instead of the entire block.

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

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

Selected Models: ChatGPT and Gemini

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Resolve inconsistency between function name and logic: (Function)

ChatGPT Output:

Certainly! There's a mismatch between the function name (`isOdd`) and its logic (`x % 2 == 0` checks for even numbers, not odd).

You can resolve it in one of **two** ways:

Option 1: Fix the Function Name (if you want to check if a number is even)

C++

```cpp
bool isEven(int x) {
    return x % 2 == 0;
}
```

Option 2: Fix the Logic (if you want to check if a number is odd)

C++

```cpp
bool isOdd(int x) {
    return x % 2 != 0;
}
```

Gemini Output:

You're right, the function name `isOdd` suggests it should return `true` if the input `x` is odd, but the logic `x % 2 == 0` checks if `x` is even.

Here are a couple of ways to resolve this inconsistency:

Option 1: Change the function name to match the logic

C++

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

This is often the simpler and clearer solution if the current logic is what you actually need.

Option 2: Change the logic to match the function name

C++

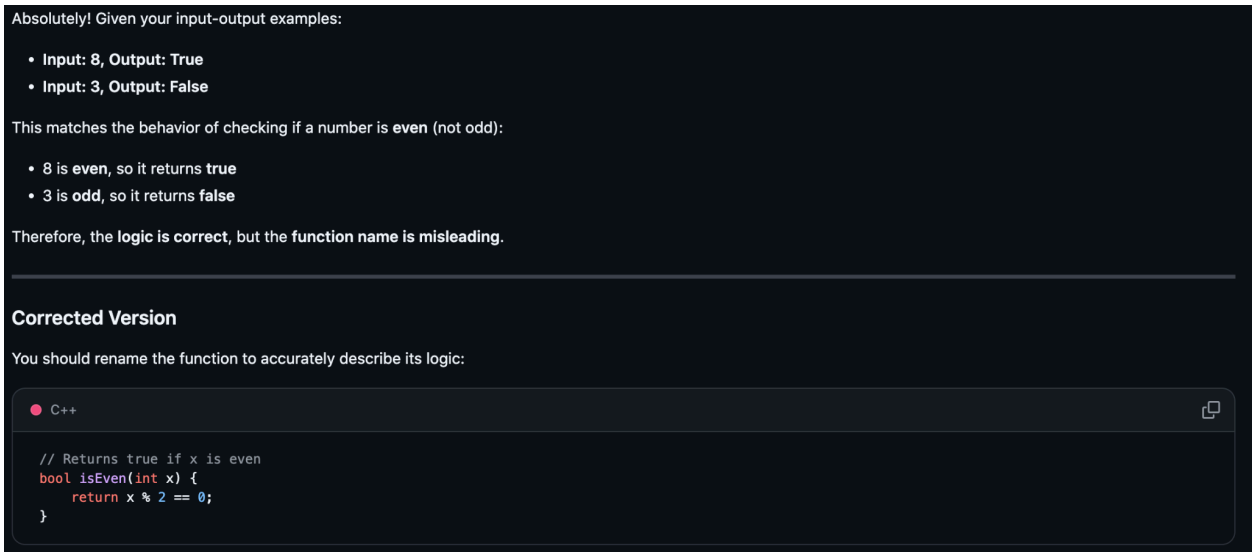
```cpp
// Returns true if x is odd
bool isOdd (int x ) {
  return x % 2 != 0; // or return x % 2 == 1;
}
```

Comment: Both ChatGPT and Gemini are able to correctly provide two options for resolving the issue, one revolving around changing the name and one around changing the logic. They both need more clarification as to whether the comment or logic should be corrected.

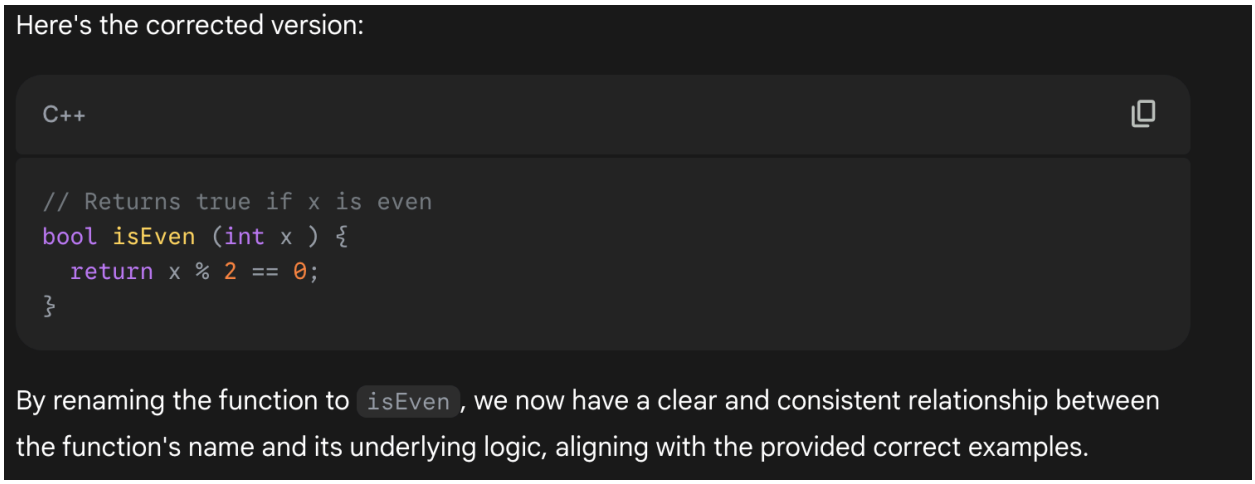
Prompting Strategy 2: Two-Shot Prompting
Prompt: Resolve inconsistency between function name and logic: (Function)
Input-Output Examples:

- Input: 8, Output: True
- Input: 3, Output: False

ChatGPT Output:



Gemini Output:



Comment: Now that examples are given, both models are able to successfully identify which correction is appropriate.

18. **Prompt Chaining: Bug Identification → Fix (JavaScript)** Identify and fix the bug:

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

Selected Models: Claude and LLaMA

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Identify and fix the bug in this JavaScript code: (code)
Claude Output:

Let me analyze this function step by step:

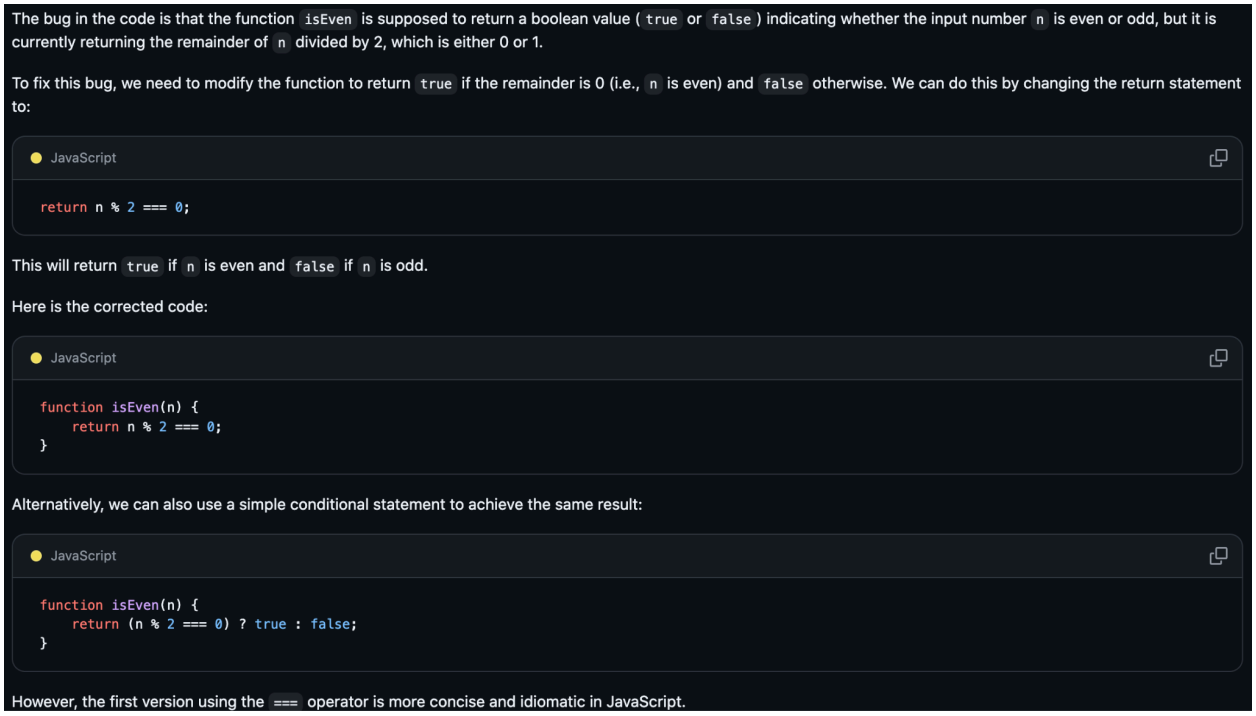
1. The function is called `isEven` which suggests it should return a boolean value (true if the number is even, false if odd).
2. The current implementation uses `n % 2`, which returns:
   - 0 when `n` is even
   - 1 when `n` is odd
3. The bug: The function returns numeric values (0 or 1) instead of boolean values (true or false) as expected from a function with "is" in its name.
4. While 0 is falsy and 1 is truthy in JavaScript, returning explicit boolean values is better practice for functions that check conditions.

Here's the fixed version:

```javascript
function isEven(n) {
  return n % 2 === 0; // Returns true for even numbers, false for odd numbers
}
```

This correction compares the result of `n % 2` with 0, returning a proper boolean value as expected from the function name.

LLaMA Output:



The bug in the code is that the function `isEven` is supposed to return a boolean value (`true` or `false`) indicating whether the input number `n` is even or odd, but it is currently returning the remainder of `n` divided by 2, which is either 0 or 1.

To fix this bug, we need to modify the function to return `true` if the remainder is 0 (i.e., `n` is even) and `false` otherwise. We can do this by changing the return statement to:

JavaScript

```
return n % 2 === 0;
```

This will return `true` if `n` is even and `false` if `n` is odd.

Here is the corrected code:

JavaScript

```
function isEven(n) {
    return n % 2 === 0;
}
```

Alternatively, we can also use a simple conditional statement to achieve the same result:

JavaScript

```
function isEven(n) {
    return (n % 2 === 0) ? true : false;
}
```

However, the first version using the `===` operator is more concise and idiomatic in JavaScript.

Comment: Both models return the same answer correctly, however Llama shows that you could also use a ternary operator to achieve the same result, although it would be less concise.

Prompting Strategy 2: Prompt Chaining
Prompt 1: Examine the following Javascript code and Identify any possible flaws or bugs. (code)
Prompt 2: Provide a solution to the bug you just identified.

Claude Prompt 1 Output:

There's a flaw in the `isEven` function. The function is supposed to check if a number is even, but it doesn't return a boolean value (true/false) as the comment suggests it should.

When you use the modulo operator (`%`) with 2, it returns:

- 0 if the number is even
- 1 if the number is odd

While 0 is falsy and 1 is truthy in JavaScript, this isn't the same as returning actual boolean values. To fix this, you should convert the result to a boolean value:

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

This way, the function explicitly returns `true` for even numbers and `false` for odd numbers, making the code more readable and predictable, especially when used in conditional statements.

Claude Prompt 2 Output:

Here's the fixed version of the `isEven` function:

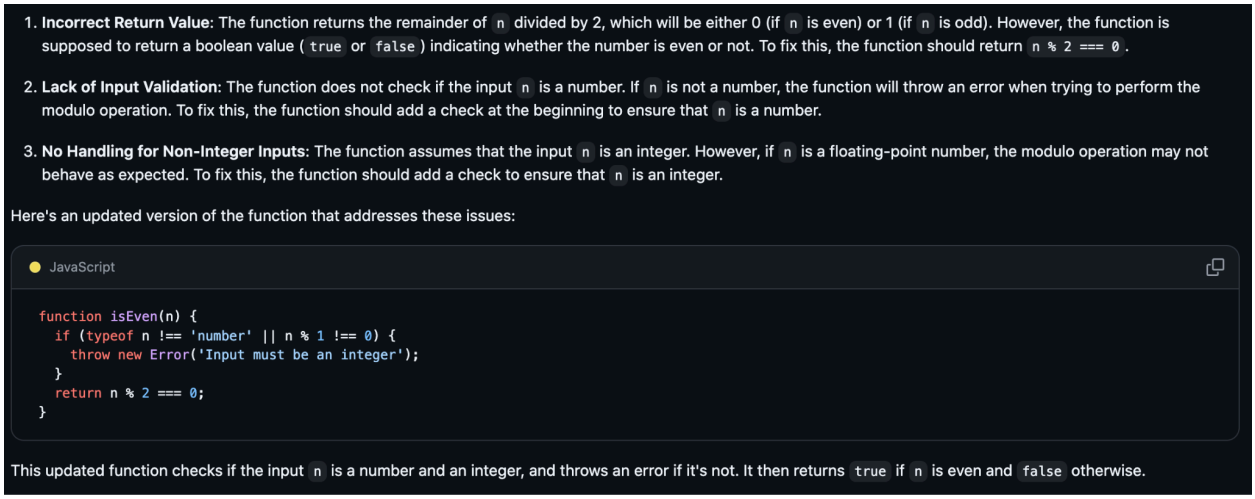
```javascript
function isEven(n) {
  return n % 2 === 0; // Returns true if even, false if odd
}
```

This solution explicitly compares the remainder to 0, returning a proper boolean value:

- When `n` is even, `n % 2` equals 0, so `n % 2 === 0` evaluates to `true`
- When `n` is odd, `n % 2` equals 1, so `n % 2 === 0` evaluates to `false`

This makes the function behave as expected for checking if a number is even, and the return value can be directly used in conditional statements without any ambiguity.

LLaMA Prompt 1 Output:



1. **Incorrect Return Value**: The function returns the remainder of `n` divided by 2, which will be either 0 (if `n` is even) or 1 (if `n` is odd). However, the function is supposed to return a boolean value (`true` or `false`) indicating whether the number is even or not. To fix this, the function should return `n % 2 === 0`.
2. **Lack of Input Validation**: The function does not check if the input `n` is a number. If `n` is not a number, the function will throw an error when trying to perform the modulo operation. To fix this, the function should add a check at the beginning to ensure that `n` is a number.
3. **No Handling for Non-Integer Inputs**: The function assumes that the input `n` is an integer. However, if `n` is a floating-point number, the modulo operation may not behave as expected. To fix this, the function should add a check to ensure that `n` is an integer.

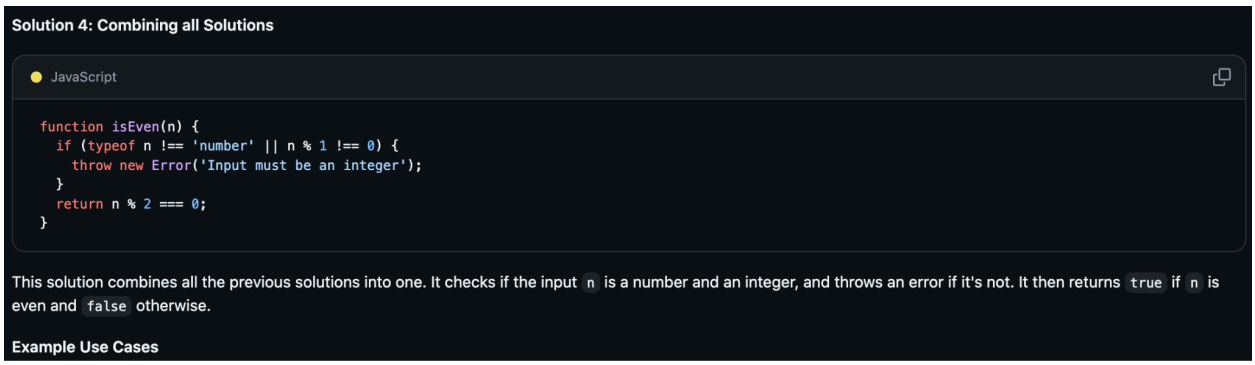
Here's an updated version of the function that addresses these issues:

JavaScript

```
function isEven(n) {
  if (typeof n !== 'number' || n % 1 !== 0) {
    throw new Error('Input must be an integer');
  }
  return n % 2 === 0;
}
```

This updated function checks if the input `n` is a number and an integer, and throws an error if it's not. It then returns `true` if `n` is even and `false` otherwise.

LLaMA Prompt 2 Output:



Solution 4: Combining all Solutions

JavaScript

```
function isEven(n) {
  if (typeof n !== 'number' || n % 1 !== 0) {
    throw new Error('Input must be an integer');
  }
  return n % 2 === 0;
}
```

This solution combines all the previous solutions into one. It checks if the input `n` is a number and an integer, and throws an error if it's not. It then returns `true` if `n` is even and `false` otherwise.

Example Use Cases

Comment: Both models jump the gun and provide a new solution upon the first prompt. I considered explicitly telling the model not to solve the problem, but I decided against it as that is rather unlikely for a person prompting the AI to actually do. Llama provides a slightly more robust solution which validates inputs to be integers.

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

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

Selected Models: ChatGPT and Claude

Prompting Strategy 1: Self-Consistency
Prompt: Decompose the high-level comment/summary into logical steps: (function)
ChatGPT Output 1:

OpenAI GPT-4.1 now

Certainly! Here's a decomposition of the high-level comment/summary into logical steps for the function:

High-level summary:
// Function that validates an input, calculates square, and returns result

Decomposed logical steps:

1. **Receive an integer input (`x`).**
2. **Validate the input:**
   - Check if `x` is less than 0 (i.e., negative).
3. **If the input is invalid (negative):**
   - Return `-1` to indicate an error or invalid input.
4. **If the input is valid (zero or positive):**
   - Calculate the square of `x` (i.e., `x * x`).
5. **Return the calculated square as the result.**

ChatGPT Output 2:

Certainly! Here's a decomposition of the high-level comment/summary into logical steps for the function:

High-level summary:
// Function that validates an input, calculates square, and returns result

Decomposed logical steps:

1. **Receive an integer input (`x`).**
2. **Check if the input is valid (i.e., non-negative):**
   - If `x` is less than 0, return `-1` to indicate invalid input.
3. **If the input is valid, calculate the square of the input (`x * x`).**
4. **Return the calculated square as the result.**

ChatGPT Output 3:

Certainly! Here's a decomposition of the high-level comment/summary into logical steps for the function:

High-level summary:
// Function that validates an input, calculates square, and returns result

Decomposed logical steps:

1. **Receive an integer input (`x`).**
2. **Validate the input:**
   - Check if `x` is less than 0 (i.e., negative).
   - If `x` is negative, return `-1` to indicate invalid input.
3. **Calculate the square:**
   - If `x` is non-negative, compute the square of `x` (i.e., `x * x`).
4. **Return the result:**
   - Return the squared value as the result.

In summary:

- Input validation
- Error handling for invalid input
- Calculation of square
- Returning the result

Claude Output 1:

I'll decompose the high-level summary into logical steps that match the implementation:

1. **Input Validation**
   - Check if the input value `x` is negative
   - If negative, return an error code (-1)
2. **Calculation**
   - Calculate the square of the input value (x * x)
3. **Return Result**
   - Return the calculated square value to the caller

These steps follow the function's implementation where it first validates that the input isn't negative, then calculates the square of the input, and finally returns the result.

Claude Output 2:

I'll decompose the high-level summary into logical steps that match the implementation:

1. **Input Validation**
   - Check if the input value `x` is negative
   - If negative, return an error code (-1)
2. **Calculation**
   - Calculate the square of the input value (x * x)
3. **Return Result**
   - Return the calculated square value to the caller

These steps follow the function's implementation where it first validates that the input isn't negative, then calculates the square of the input, and finally returns the result.

Claude Output 3:

I'll decompose the high-level summary into logical steps that match the implementation:

1. **Input Validation**
   - Check if the input value `x` is negative
   - If negative, return an error code (-1)
2. **Calculation**
   - Calculate the square of the input value (x * x)
3. **Return Result**
   - Return the calculated square value to the caller

These steps follow the function's implementation where it first validates that the input isn't negative, then calculates the square of the input, and finally returns the result.

Comment: Claude was very consistent with such a simple question, and created the same result every single time even when re-prompted. This might also be a sign of Claude's lower estimated temperature at .6. When ChatGPT was prompted to re-run, it returned slightly different responses each time, but the responses were materially the same.

Prompting Strategy 2: Chain-of-Thought
Prompt: Explain in a step by step process your thoughts before coming to a final answer. Decompose the high-level comment/summary into logical steps: (function)

ChatGPT Output:

Step 1: Read and Understand the High-Level Comment

The comment says:
// Function that validates an input, calculates square, and returns result

This tells me the function does three main things:

- Validates the input
- Calculates the square
- Returns the result

Step 2: Examine the Function Implementation

The function is:

C++

```cpp
int process(int x) {
    if (x < 0) return -1;
    return x * x;
}
```

I see that:

- The function takes an integer `x` as input.
- It checks if `x` is negative. If so, it returns `-1`.
- Otherwise, it returns `x * x`.

Step 3: Map Implementation to the High-Level Steps

Now, I want to break down the comment into more detailed, logical steps that match the code.

a. Input Reception

- The function receives an integer input `x`.

b. Input Validation

- The function checks if the input is valid. In this context, "valid" means "not negative".
- If the input is invalid (i.e., negative), it returns `-1` as an error code.

c. Calculation

- If the input is valid (i.e., zero or positive), the function calculates the square of `x`.

d. Return Result

- The function returns either the error code or the calculated square.

Step 4: Write Out the Logical Steps

Based on the above, the logical steps are:

1. **Receive the input integer `x`.**
2. **Check if `x` is negative:**
   - If yes, return `-1` to indicate invalid input.
3. **If `x` is non-negative:**
   - Calculate the square of `x` (`x * x`).
   - Return the squared value.

Step 5: Final Decomposition

Decomposed logical steps:

1. Accept an integer input.
2. Validate the input by checking if it is negative.
3. If the input is negative, return `-1` as an error code.
4. If the input is non-negative, calculate its square.
5. Return the result of the calculation.

Claude Output:

Let me think step by step to decompose this high-level comment/summary into logical steps:

1. First, I need to understand the function's purpose from the comment: "Function that validates an input, calculates square, and returns result".
2. Then, I need to analyze the actual code implementation:

```
int process(int x) {
  if (x < 0) return -1;
  return x * x;
}
```

3. The code accepts an integer parameter `x`.
4. It performs a validation check: `if (x < 0) return -1;`
   - This checks if x is negative
   - If x is negative, it returns -1 (likely an error code)
   - This corresponds to the "validates an input" part of the comment
5. If x passes validation (i.e., x is not negative), it calculates `x * x`
   - This corresponds to the "calculates square" part of the comment
6. The function then returns this calculated result
   - This corresponds to the "returns result" part of the comment
7. Now I can formalize these steps into a logical breakdown that matches the function's implementation:

Final Answer:

1. **Input Validation**
   - Check if the input value `x` is negative
   - If negative, return an error code (-1)
2. **Calculate Square**
   - Multiply the input value by itself (x * x)
3. **Return Result**
   - Return the calculated square value to the caller

Comment: This problem is likely also to simple to have realistic gains in performance through Chain-of-Thought reasoning. The most interesting difference in the output was how much more thorough ChatGPT was when prompted, but also somewhat redundant.

Whereas Claude accomplishes the request but keeps each step rather short while still being understandable.

20. **Purpose Inference → Completion (Python)** Complete the function based on intent:

```
def calculate_average(scores):
    total = 0
    # TODO: Complete to return average
    pass
```

Selected Models: Claude and Gemini

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Complete the function based on intent: (Function)
Claude Output:

Here's the completed function to calculate the average of scores:

```python
def calculate_average(scores):
    total = 0
    for score in scores:
        total += score
    if len(scores) > 0:
        return total / len(scores)
    else:
        return 0  # Return 0 for empty list to avoid division by zero
```

This function:

1. Sums all values in the scores list using a for loop
2. Divides the total by the number of scores to get the average
3. Handles the edge case of an empty list by returning 0 instead of causing a division by zero error

Would you like me to explain any part of this implementation?

Gemini Output:

Python

```
def calculate_average(scores):
    total = 0
    if not scores:
        return 0
    for score in scores:
        total += score
    average = total / len(scores)
    return average
```

Comment: Gemini is very to-the-point, while Claude explains the code a bit more. Gemini's code handles the exception of having an empty list sooner, making it slightly more efficient.

Prompting Strategy 2: Three-Shot Prompting
Prompt: Complete the function based on intent:

Python

```
def calculate_average(scores):
    total = 0
    # TODO: Complete to return average
    pass
```

Here are three examples of how the function should work:
Example 1:
Python

```
print(calculate_average([80, 90, 100]))  # Output: 90.0
```

Example 2:
Python

```
print(calculate_average([50, 75]))  # Output: 62.5
```

Example 3:
Python

```
print(calculate_average([10, 20, 30, 40]))  # Output: 25.0
```

Claude Output:

```
def calculate_average(scores):
    total = 0

    # Calculate the sum of all scores
    for score in scores:
        total += score
```

```
    # Calculate the average
    average = total / len(scores)

    return average
```

Gemini Output:

```
def calculate_average(scores):
    total = sum(scores)
    if scores:
        return total / len(scores)
    else:
        return 0.0
```

Comment: Gemini provides slightly more robust code which also utilizes built-in functions like sum().

21. **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.
Bonus Question — Worth up to 2.5 points toward the total grade

```python
# 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()
```

Selected Models: LLaMA and ChatGPT

Prompting Strategy 1: Zero-Shot Prompting
Prompt: Analyze the following utility script. Identify and highlight any possible logical issues or design flaws. Refactor the file for better readability, correctness and safety. (File)
ChatGPT Output:

```
import csv
import sys
from typing import List, Tuple

def read_csv(filepath: str) -> List[List[str]]:
    """
    Reads a CSV file and returns its content as a list of rows.
    Each row is a list of strings.
    """
    try:
        with open(filepath, 'r', newline='') as f:
            reader = csv.reader(f)
            data = [row for row in reader]
        if not data:
            raise ValueError("CSV file is empty.")
        return data
    except FileNotFoundError:
        print(f"Error: File '{filepath}' not found.")
        sys.exit(1)
    except Exception as e:
        print(f"Error reading CSV file: {e}")
        sys.exit(1)

def summarize_column(data: List[List[str]], index: int) -> Tuple[float, float]:
    """
    Summarizes a numeric column in the CSV data.
    Skips the header row and any rows with invalid or missing data.
    Returns the total and average of the column.
    """
    values = []
    for row_num, row in enumerate(data[1:], start=2):  # start=2 for line number
        try:
            value = float(row[index])
            values.append(value)
        except (IndexError, ValueError):
            print(f"Warning: Skipping invalid or missing value at line {row_num}.")
    if not values:
        print("Error: No valid numeric data found in the specified column.")
        sys.exit(1)
    total = sum(values)
    avg = total / len(values)
    return total, avg

def main():
```

```
    if len(sys.argv) != 3:
        print("Usage: python utils.py <csv_file> <column_index>")
        sys.exit(1)
    filepath = sys.argv[1]
    try:
        col_index = int(sys.argv[2])
        if col_index < 0:
            raise ValueError
    except ValueError:
        print("Error: Column index must be a non-negative integer.")
        sys.exit(1)

    data = read_csv(filepath)
    header = data[0]
    if col_index >= len(header):
        print(f"Error: Column index {col_index} is out of range. CSV has {len(header)}
columns.")
        sys.exit(1)

    total, avg = summarize_column(data, col_index)
    print(f"Total: {total}")
    print(f"Average: {avg}")

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

LLaMA Output:

```
import csv
from typing import List, Tuple

def read_csv_file(file_path: str) -> List[List[str]]:
    """
    Reads a CSV file and returns its contents as a list of lists.

    Args:
        file_path (str): The path to the CSV file.

    Returns:
        List[List[str]]: The contents of the CSV file.
    """
    try:
        with open(file_path, 'r') as file:
            csv_reader = csv.reader(file)
            data = list(csv_reader)
```

```
            return data
    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
        return []
    except Exception as e:
        print(f"An error occurred: {e}")
        return []

def summarize_column(data: List[List[str]], index: int) -> Tuple[float, float]:
    """
    Calculates the total and average of a column in the data.

    Args:
        data (List[List[str]]): The data to summarize.
        index (int): The index of the column to summarize.

    Returns:
        Tuple[float, float]: A tuple containing the total and average.
    """
    if index < 0 or index >= len(data[0]):
        raise ValueError("Invalid column index")

    values = [float(row[index]) for row in data[1:]]  # skip header
    if not values:
        raise ValueError("No values found in the column")

    total = sum(values)
    avg = total / len(values)
    return total, avg

def print_summary(total: float, avg: float) -> None:
    """
    Prints the total and average to the console.

    Args:
        total (float): The total value.
        avg (float): The average value.
    """
    print(f"Total: {total}")
    print(f"Average: {avg}")

def main() -> None:
    file_path = 'data.csv'
    data = read_csv_file(file_path)
```

```
    if not data:
        return

    column_index = 1
    try:
        total, avg = summarize_column(data, column_index)
        print_summary(total, avg)
    except ValueError as e:
        print(f"Error: {e}")

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

Comment: Outputs once again both meet the guidelines, but Llama takes more liberty than ChatGPT in terms of changes that it makes, renaming functions and creating new functions. ChatGPT is a safer refactoring with error handling and as minimal changes as possible, while Llama tries to be more modular and reusable, but can no longer function in the command line.

Prompting Strategy 2: Prompt Chaining
Prompt 1: You are an expert in Python programming and are meant to assist me in my python-related tasks. It is incredibly important that you double check your work and test if it functions correctly.
Prompt 2: Analyze the following utility script. Identify and highlight any possible logical issues or design flaws. (File)
Prompt 3: Refactor the file for better readability, correctness and safety.

ChatGPT Output:

```
import csv
from typing import List, Tuple

def read_csv_file(file_path: str) -> List[List[str]]:
    """
    Reads a CSV file and returns its contents as a list of lists.

    Args:
        file_path (str): The path to the CSV file.

    Returns:
        List[List[str]]: The contents of the CSV file.
    """
    try:
        with open(file_path, 'r') as file:
            csv_reader = csv.reader(file)
```

```
            data = list(csv_reader)
            return data
    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
        return []
    except Exception as e:
        print(f"An error occurred: {e}")
        return []

def get_column_values(data: List[List[str]], column_index: int) -> List[float]:
    """
    Extracts the values from a specific column in the data.

    Args:
        data (List[List[str]]): The data to extract values from.
        column_index (int): The index of the column to extract values from.

    Returns:
        List[float]: The values from the specified column.
    """
    if column_index < 0 or column_index >= len(data[0]):
        raise ValueError("Invalid column index")

    values = [float(row[column_index]) for row in data[1:]]  # skip header
    return values

def calculate_total_and_average(values: List[float]) -> Tuple[float, float]:
    """
    Calculates the total and average of a list of values.

    Args:
        values (List[float]): The values to calculate the total and average for.

    Returns:
        Tuple[float, float]: A tuple containing the total and average.
    """
    if not values:
        raise ValueError("Cannot calculate total and average for empty list")

    total = sum(values)
    average = total / len(values)
    return total, average

def main():
```

```
    file_path = 'data.csv'
    data = read_csv_file(file_path)
    if not data:
        return

    column_index = 1
    values = get_column_values(data, column_index)
    if not values:
        print("No values found in the specified column.")
        return

    total, average = calculate_total_and_average(values)
    print(f"Total: {total}")
    print(f"Average: {average}")

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

LLaMA Output:

```
import csv
import sys
from typing import List, Tuple

def read_csv(filepath: str) -> List[List[str]]:
    """
    Reads a CSV file and returns its content as a list of rows.
    Each row is a list of strings.
    """
    try:
        with open(filepath, 'r', newline='') as f:
            reader = csv.reader(f)
            data = [row for row in reader]
        if not data:
            raise ValueError("CSV file is empty.")
        return data
    except FileNotFoundError:
        print(f"Error: File '{filepath}' not found.")
        sys.exit(1)
    except Exception as e:
        print(f"Error reading CSV file: {e}")
        sys.exit(1)

def summarize_column(data: List[List[str]], index: int) -> Tuple[float, float]:
    """
```

```
    Summarizes a numeric column in the CSV data.
    Skips the header row and any rows with invalid or missing data.
    Returns the total and average of the column.
    """
    values = []
    for row_num, row in enumerate(data[1:], start=2):  # start=2 for line number
        try:
            value = float(row[index])
            values.append(value)
        except (IndexError, ValueError):
            print(f"Warning: Skipping invalid or missing value at line {row_num}.")
    if not values:
        print("Error: No valid numeric data found in the specified column.")
        sys.exit(1)
    total = sum(values)
    avg = total / len(values)
    return total, avg

def main():
    if len(sys.argv) != 3:
        print("Usage: python utils.py <csv_file> <column_index>")
        sys.exit(1)
    filepath = sys.argv[1]
    try:
        col_index = int(sys.argv[2])
        if col_index < 0:
            raise ValueError
    except ValueError:
        print("Error: Column index must be a non-negative integer.")
        sys.exit(1)

    data = read_csv(filepath)
    header = data[0]
    if col_index >= len(header):
        print(f"Error: Column index {col_index} is out of range. CSV has {len(header)}
columns.")
        sys.exit(1)

    total, avg = summarize_column(data, col_index)
    print(f"Total: {total}")
    print(f"Average: {avg}")

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

Comment: Llama created a more flexible and modular script by using sys.argv. Llama's also checks inputs for if they are empty. ChatGPT will raise errors if the type conversion fails, while Llama will skip the row and print a warning message.

22. **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.
Bonus Question — Worth up to 2.5 points toward the total grade

```
# 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()
```

Selected Models: Claude and Gemini

Prompting Strategy 1: Chain-of-Thought
Prompt: Explain in a step by step process your thoughts before coming to a final answer. Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly. (File)
Claude Output:

The script is now complete! I've implemented the `clean_line` function to:

1. Convert the line to lowercase
2. Remove all punctuation characters using Python's `string.punctuation` and the `translate` method

For the bonus question, I've considered these aspects of robustness:

- The script uses `get()` with a default value to handle new words gracefully
- The `translate` method efficiently removes all punctuation in one pass
- The function properly converts text to lowercase before processing
- Error handling could be improved by adding try/except blocks around the file operations

Would you like me to explain any part of the code in more detail or enhance it with additional error handling?

Gemini Output:

Explanation of Changes:

1. `clean_line(line)` **Function:**
    - `line = line.strip().lower()`: This line first removes any leading or trailing whitespace from the input `line` using `.strip()` and then converts the entire line to lowercase using `.lower()`. This ensures that words like "The" and "the" are counted as the same word.
    - `line = line.translate(str.maketrans('', '', string.punctuation))`: This is a robust way to remove punctuation.
        - `string.punctuation` provides a string containing all standard punctuation marks.
        - `str.maketrans('', '', string.punctuation)` creates a translation table that maps each punctuation character to `None`, effectively deleting them during the translation.
        - `line.translate(...)` applies this translation table to the `line`.
2. `count_words(lines)` **Function:**
    - `if word:`: This check is added within the inner loop to ensure that empty strings (which might result from multiple spaces after cleaning) are not counted as words.

3. `main()` **Function:**
    - **Error Handling (`try...except` block):** A `try...except` block is added to handle potential errors during file processing:
        - `FileNotFoundError`: Catches the specific error that occurs if the `input.txt` file is not found at the specified path. It prints a user-friendly error message.
        - `Exception as e`: Catches any other potential exceptions that might occur during file reading or processing and prints a general error message along with the specific error.

To run this script:

1. Save the code as `file_processor.py`.
2. Create a text file named `input.txt` in the same directory as the script. Populate `input.txt` with some text containing punctuation and mixed-case words. For example:

```
This is a sample text, with some punctuation!
It has multiple lines and The word "the" appears twice.
```

3. Run the script from your terminal: `python file_processor.py`

Expected Output (for the example `input.txt`):

Comment: Both Models outputted a correct solution to the problem before any explanation, the main difference was in the thoroughness of explanation. Gemini was more thorough and provided an expected output given a sample txt file.

Prompting Strategy 2: Prompt Chaining
Prompt 1: You are an expert in Python programming and are meant to assist me in my python-related tasks. It is incredibly important that you double check your work and test if it functions correctly.
Prompt 2: I have a skeleton of a file-processing script which is meant to clean each line, remove punctuation, and count word frequencies correctly.
Prompt 3: Please complete this skeleton File: (File)
Claude Output:

```
# file_processor.py - Script for processing text files
import string

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

```
def clean_line(line):
    # Remove punctuation and make lowercase
    line = line.lower()
    # Remove all punctuation characters
    line = line.translate(str.maketrans('', '', string.punctuation))
    return line

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()
```

Gemini Output:
```
# file_processor.py - Incomplete script for processing text files
import string

def load_file(filepath):
    try:
        with open(filepath, 'r') as f:
            return f.readlines()
    except FileNotFoundError:
        print(f"Error: File not found at '{filepath}'")
        return []

def clean_line(line):
    # Remove punctuation
    line = line.translate(str.maketrans('', '', string.punctuation))
    # Make lowercase and remove leading/trailing whitespace
    return line.strip().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(filepath)
    if lines:
        counts = count_words(lines)
        for word, count in sorted(counts.items()):
            print(f"{word}: {count}")

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

Comment: The two versions are quite similar, however gemini has error handling if the file is missing, as well as checks if the list is empty. Both ultimately meet the requirements of the prompt.