

GenAI for Software Development: Assignment 3

Benjamin Normann
benormann@wm.edu

Introduction

This report compares language model performance across 22 software development tasks. For each task, we present a representative prompt and response (either zero-shot or few-shot), with a brief analysis of output quality. Two models are considered: OpenAI's GPT-4o and Meta's Llama 3.1-405B-Instruct. The impact of prompt engineering (zero-shot vs. few-shot) is also evaluated where applicable. A summary at the end provides an overall assessment of model performance and prompting strategies.

Task 1: Code Summarization (Java)

Prompting Strategy 1: Zero-shot Prompt:

Summarize the functionality of the following method:

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

GPT-4o Response:

The method `countWordFrequency` takes a list of strings (`words`) as input and returns a map (`Map<String, Integer>`) that represents the frequency of each word in the list. Here's how it works:

1. It initializes an empty `HashMap` called `freqMap` to store words as keys and their respective frequencies as values.
2. It iterates through each word in the input list (`words`).
3. For each word, it checks if the word already exists in the map:
 - If it exists, it increments the count (value) by 1.
 - If it does not exist, it adds the word to the map with an initial count of 1.
4. Finally, it returns the `freqMap` containing the word-frequency pairs.

In summary, the method calculates and returns a frequency distribution of words from the input list.

Response time: 3.85 seconds

Llama-3.1 Response:

This method, `countWordFrequency`, takes a list of words as input and returns a map where the keys are the unique words and the values are their corresponding frequencies.

Here's a step-by-step breakdown:

1. It initializes an empty `HashMap`freqMap`` to store the word frequencies.
2. It then iterates over each word in the input list.
3. For each word, it checks if the word is already in the map using ``freqMap.getDefault(word, 0)``. If the word is present, it returns its current frequency; otherwise, it returns 0.
4. It increments the frequency by 1 and updates the map with the new frequency using ``freqMap.put(word, ...)``.
5. Finally, it returns the map containing the frequency of each word.

Example usage:

```
```java
List<String> words = Arrays.asList("apple", "banana", "apple", "orange", "banana", "banana");
Map<String, Integer> freqMap = countWordFrequency(words);
// freqMap would contain: {"apple": 2, "banana": 3, "orange": 1}
```
```

Response time: 7.58 seconds

Few-shot Prompt:

Summarize the functionality of the following method:

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

Example 1:

This method counts the frequency of each word in a list and returns a map where keys are words and values are their occurrence counts.

Example 2:

This method creates a frequency map of words. It iterates through a list of strings, incrementing the count for each word in a `HashMap`, and returns the resulting frequency map.

GPT-4o Response:

The method ``countWordFrequency`` takes a list of strings (``words``) as input and returns a map where each unique word is a key, and its corresponding value is the number of times that word appears in the list. It works as follows:

1. Initializes an empty ``HashMap`` (``freqMap``) to store word frequencies.
2. Iterates through each word in the input list.
3. For each word, it updates the frequency in the map by using ``getDefault`` to retrieve the current count (defaulting to 0 if the word is not already in the map), then increments it by 1.
4. After processing all words, it returns the ``freqMap`` containing the word-frequency pairs.

Key Points:

- Handles duplicate words by incrementing their count in the map.
- Uses ``getDefault`` to simplify handling of new words.
- Returns a ``HashMap`` with word-frequency mappings.

Example Input and Output:

Input: `["apple", "banana", "apple", "orange", "banana", "banana"]`

Output: `{"apple":2, "banana":3, "orange":1}`

Response time: 4.08 seconds

Llama-3.1 Response:

The provided method, `countWordFrequency`, takes a list of words as input and returns a map where each word is a key, and its corresponding value is the frequency of that word in the list. It achieves this by iterating over each word in the list and incrementing its count in the map. If the word is not already in the map, it defaults to a count of 0 before incrementing.

Both example descriptions accurately summarize the functionality of the method. The first example focuses on the purpose of the method, while the second example provides a more detailed explanation of how it works.

Here's a more concise version of the descriptions:

"This method returns a map of word frequencies from a given list of words."

or

"This method creates a frequency map of words by iterating over a list and counting each word's occurrence."

Response time: 5.97 seconds

Analysis: Both GPT-4o and Llama accurately summarized the Java method for counting word frequencies. GPT-4o delivered a step-by-step summary, and Llama also provided a step-by-step explanation, but with an example. Few-shot prompting enhanced formatting, with GPT-4o adding Key Points and an example, Llama analyzed the examples provided for conciseness (a strange response). GPT-4o was faster (3.85s vs. 7.58s in zero-shot)

Comparison: Choose GPT-4o for quick, readable summaries or Llama for detailed breakdowns. GPT-4o generated a more relevant answer than Llama with few-shot prompting.

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

Prompting Strategy 1: Zero-shot Prompt:

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

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

GPT-4o Response:

The function `sum_range` adds all numbers from `start` to `end`, but there is an **off-by-one error** because the `range(start, end)` function in Python excludes the `end` value. If the intention is to include both `start` and `end` in the summation, the off-by-one error can be fixed by changing `range(start, end)` to `range(start, end + 1)`.

Here's the corrected code:

```
```python
def sum_range(start, end):
 total = 0
 for i in range(start, end + 1): # Add 1 to 'end' to include it in the range
 total += i
 return total
```
```

Example:

```
```python
print(sum_range(1, 5)) # Correct output: 1 + 2 + 3 + 4 + 5 = 15
```
```

If you intend for the original behavior (excluding the `end` value) to be correct, then no changes are necessary, but in most cases, summing a range typically includes both endpoints.

Response time: 21.14 seconds

Llama-3.1 Response:

The off-by-one error in the function is due to the exclusive nature of the `end` parameter in the `range()` function in Python. This means that the `end` value is not included in the range.

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

```
```python
def sum_range(start, end):
 total = 0
 for i in range(start, end + 1): # Include the end value by adding 1
 total += i
 return total
```
```

Alternatively, you can use the `sum()` function in combination with `range()` to simplify the function:

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

This version achieves the same result in a more concise way.

Response time: 6.11 seconds

Few-shot Prompt:

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

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

Example 1:

The function has an off-by-one error because `range(start, end)` excludes the 'end' value. To include it, change to `range(start, end+1)`:

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

Example 2:

The issue is that `range(start, end)` doesn't include 'end' itself. Fix by using `range(start, end+1)` to include all numbers from start to end inclusive:

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

GPT-4o Response:

Both examples correctly identified the off-by-one error in the original function and provided the same fix. Here's a summary of the issue and the solution:

Issue:

The Python `range(start, end)` function generates numbers starting from `start` and stops **before** `end`. This means the `end` value is excluded from the range. If the goal is to include `end` in the sum, the function needs to account for this.

Fix:

To include the `end` value in the range, adjust the `end` parameter in the `range()` function to `end + 1`. This ensures that the range includes all integers from `start` to `end` (inclusive).

Corrected Function:

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

Example Behavior:

Input:

```
```python
print(sum_range(1, 5)) # Sum from 1 to 5
```
```

Output:

```
15 # (1 + 2 + 3 + 4 + 5)
```

This correction ensures the function behaves as expected, summing all integers from `start` to `end`, inclusive.

Response time: 11.23 seconds

Llama-3.1 Response:

The off-by-one error in the function is due to the fact that the `range()` function in Python is exclusive of the end value. This means that the loop will iterate up to but not including the `end` value.

To fix this error, you can modify the `range()` function to `range(start, end+1)`, which will include the `end` value in the iteration.

Here is the corrected code:

```
```python
def sum_range(start, end):
 """
 Returns the sum of all integers in the range from start to end (inclusive).
 Args:
 start (int): The start of the range.
 end (int): The end of the range.

 Returns:
 int: The sum of all integers in the range.
 """
 total = 0
 for i in range(start, end+1):
 total += i
 return total
```
```

Example usage:

```
```python
print(sum_range(1, 5)) # Output: 15 (1+2+3+4+5)
print(sum_range(10, 20)) # Output: 155 (10+11+12+13+14+15+16+17+18+19+20)
```
```

Alternatively, you can use the formula for the sum of an arithmetic series to simplify the function:

```
```python
def sum_range(start, end):
 """
 Returns the sum of all integers in the range from start to end (inclusive).
 Args:
 start (int): The start of the range.
 end (int): The end of the range.

 Returns:
 int: The sum of all integers in the range.
 """
 return (end + start) * (end - start + 1) // 2
```
```

This version of the function achieves the same result without the need for a loop.

Response time: 613.28 seconds

Analysis: Both models fixed the off-by-one error by adjusting the range to range(start, end + 1). GPT-4o's few-shot response included an example and output (e.g., sum_range(1, 5) = 15), while Llama suggested an alternative simplified formula utilizing sum in conjunction with range again for conciseness, but this time relevant. Few-shot prompting improved clarity and for Llama it added examples and more descriptive commenting at the cost of a ridiculous response time.

Comparison: Use GPT-4o for straightforward fixes with examples or Llama for optimized solutions.

Task 3: Bug Classification (C++)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

```
The bug in the provided C++ function can be classified as Returning a pointer to a local variable.  
This is a lifetime issue or a dangling pointer bug.  
#### Why is it a bug?  
In C++, local variables (such as `arr` in this case) are stored on the stack and are automatically destroyed when the function returns. When the `getArray` function returns, the memory allocated for the `arr` array is deallocated, and the pointer `arr` becomes a dangling pointer. Accessing the memory through this pointer after the function returns leads to undefined behavior.  
#### Key Problems:  
1. Lifetime Issue: The array `arr` is a local variable, and its lifetime ends when the function exits.  
2. Undefined Behavior: Returning a pointer to a deallocated memory location results in undefined behavior if the caller tries to dereference or use the returned pointer.  
#### How to Fix It?  
There are several correct ways to fix this issue, depending on the desired semantics:  
##### Option 1: Use `new` to allocate memory on the heap  
Allocate the array dynamically on the heap, ensuring it remains valid after the function returns:  
```cpp  
int * getArray(int size) {
 int *arr = new int[size]; // Allocate on heap
 return arr; // Caller must delete[] the array to avoid memory leak
}
...
However, this approach places the responsibility of freeing the allocated memory (`delete[]`) on the caller.
Option 2: Use `std::vector` for automatic memory management
Instead of raw pointers and dynamic allocation, use `std::vector`, which manages memory automatically:
```cpp  
#include <vector>
```

```
std::vector<int> getArray(int size) {
    std::vector<int> arr(size); // Allocate using vector
    return arr; // Return by value (efficient copy due to move semantics)
}
...
```

This is the recommended approach in modern C++.

Option 3: Pass the array as an argument

Modify the function to accept an array as a parameter, avoiding the need to allocate memory inside the function:

```
```cpp
void getArray(int size, int arr[]) {
 for (int i = 0; i < size; ++i) {
 arr[i] = 0; // Example initialization
 }
}
...
```

This avoids the problem of returning a pointer entirely.

### Conclusion

The type of bug is a **"dangling pointer issue"** caused by returning a pointer to a local (stack-allocated) variable. Using modern C++ practices such as `std::vector` is the safest and most robust way to fix this issue.

Response time: 9.39 seconds

Llama-3.1 Response:

The bug in the given C++ function is a classic example of a "Dangling Pointer" bug.

A dangling pointer is a pointer that points to memory that has already been deallocated or reused. In this case, the function `getArray` returns a pointer to a local variable `arr`, which is allocated on the stack. When the function returns, the stack frame is destroyed, and the memory occupied by `arr` is deallocated. However, the returned pointer still points to the deallocated memory location, which can cause undefined behavior when the caller tries to access the memory through the returned pointer.

To fix this bug, you can dynamically allocate memory for the array using `new` and `delete`, or use a smart pointer like `std::unique_ptr` or `std::shared_ptr` to manage the memory. Alternatively, you can use a container class like `std::vector` to avoid manual memory management altogether.

Here's an example of how to fix the bug using dynamic memory allocation:

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

// Don't forget to delete the array when you're done with it
int main() {
    int* arr = getArray(10);
    // Use the array...
    delete[] arr;
    return 0;
}
...
```


Or using a smart pointer:

```
```cpp
#include <memory>
std::unique_ptr<int[]> getArray(int size) {
 return std::make_unique<int[]>(size);
}

int main() {
 auto arr = getArray(10);
 // Use the array...
 return 0;
}
```
```

In the latter example, the `std::unique_ptr` will automatically delete the array when it goes out of scope, so you don't need to worry about manual memory management.

Response time: 13.78 seconds

Few-shot Prompt:

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

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

Example 1:

This function contains a dangling pointer bug. It returns a pointer to a local array variable that will be deallocated when the function returns, leading to undefined behavior when the returned pointer is used.

Example 2:

The bug is a stack memory return bug. The function creates a local array on the stack and returns its address. When the function exits, the stack memory is reclaimed, making the returned pointer invalid and causing a dangling pointer issue.

GPT-4o Response:

The bug in the provided C++ function can be classified as **both** a **dangling pointer bug** and a **stack memory return bug**, as these terms describe the same underlying issue from slightly different perspectives.

Explanation:

1. **Dangling Pointer Bug**:

- A dangling pointer occurs when a pointer refers to memory that has been deallocated or is no longer valid.

- In the given code, the array `arr` is a **local variable**. Once the function `getArray` exits, the memory allocated for `arr` (on the stack) is no longer valid. Returning a pointer to this array (`return arr;`) creates a dangling pointer, as the caller of `getArray` will have a pointer to invalid memory.

2. **Stack Memory Return Bug**:

- The array `arr` is allocated on the **stack**, and stack memory is automatically reclaimed when the function exits.

- By returning a pointer to this memory, the function is effectively returning a pointer to memory that has been reclaimed, which is a violation of proper memory management and leads to undefined behavior.

Why Both Terms Apply:

- "Dangling pointer bug" focuses on the fact that the returned pointer refers to invalid memory.
- "Stack memory return bug" focuses on the specific cause of the problem: returning the address of a local stack-allocated variable.

Common Fixes:

To avoid this issue, you can:

1. Allocate the array dynamically using `new`:

```
```cpp
int * getArray(int size) {
 int *arr = new int[size];
 return arr; // Caller must free this memory using delete[]
}
...
```
```

2. Use a `std::vector` to manage memory safely:

```
```cpp
std::vector<int> getArray(int size) {
 return std::vector<int>(size);
}
...
```
```

3. Pass an array as an argument to the function:

```
```cpp
void getArray(int *arr, int size) {
 // Fill arr with values
}
...
```
```

Each of these fixes avoids returning a pointer to invalid memory, resolving both the dangling pointer and stack memory return issues.

Response time: 6.12 seconds

Llama-3.1 Response:

The bug in the given C++ function can be classified into two categories, but they essentially describe the same issue:

1. ****Dangling Pointer Bug****: This occurs when a function returns a pointer to a local variable that is no longer valid after the function returns. In this case, the local array `arr` is deallocated when the function `getArray` returns, leaving the returned pointer pointing to invalid memory.
2. ****Stack Memory Return Bug**** or ****Returning a Pointer to a Local Variable Bug****: This bug occurs when a function returns a pointer to a local variable that is allocated on the stack. When the function exits, the stack memory is reclaimed, making the returned pointer invalid.

Both categorizations describe the same underlying issue: the function is returning a pointer to memory that is no longer valid after the function returns, leading to undefined behavior when the returned pointer is used.

It's worth noting that this bug can be fixed by dynamically allocating memory for the array using `new` or `malloc`, or by using a container like `std::vector` that manages memory automatically. For example:

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

```
}  
...
```

However, it's essential to remember to `delete[]` the allocated memory when it's no longer needed to avoid memory leaks. A better approach would be to use a smart pointer like `std::unique_ptr` or a container like `std::vector` to manage memory automatically.

Response time: 10.44 seconds

Analysis: Both models correctly identified the dangling pointer bug. GPT-4o offered a structured explanation section (e.g., "Why is it a bug?") and 3 alternative options for fixing the bug (heap, smart pointers, & parameter), while Llama was less clear in its explanations (i.e. deallocation responsibility ambiguity), but also included the use of smart pointers like `std::unique_ptr` and use of the heap, but didn't include passing as a parameter. Prompt type had little effect due to consistent performance across zero-shot and few-shot scenarios.

Comparison: Opt for GPT-4o for clear explanations and complete list of alternatives. Llama did not provide any additional insights.

Task 4: Generating Email Validators (Python + Regex)

Prompting Strategy 1: Zero-shot Prompt:

```
Complete the function using regex to validate basic email addresses:  
def is_valid_email(email):  
    # TODO: Complete using regex  
    pass
```

GPT-4o Response:

```
There's a simple implementation of the `is_valid_email` function using regex to validate basic email addresses:  
```python  
import re
def is_valid_email(email):
 # Define the regex pattern for a basic email validation
 email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
 # Use re.match to check if the email matches the pattern
 return bool(re.match(email_pattern, email))
```  
  
### Explanation of the Regular Expression:  
- `^`: Ensures the pattern starts at the beginning of the string.  
- `[a-zA-Z0-9._%+-]+`: Matches the local part of the email (before the `@` symbol). It allows alphanumeric characters, dots (`.`), underscores (`_`), percent signs (`%`), plus signs (`+`), and hyphens (`-`).  
- `@`: Matches the `@` symbol.  
- `[a-zA-Z0-9.-]+`: Matches the domain name (after the `@` symbol). It allows alphanumeric characters, dots (`.`), and hyphens (`-`), but does not allow consecutive dots.
```

- `\.`: Matches the dot `.` separating the domain name and the top-level domain (TLD).
- `[a-zA-Z]{2,}`: Matches the top-level domain (e.g., `.com`, `.org`, `.net`). It requires at least 2 alphabetic characters.
- `$`: Ensures the pattern ends at the end of the string.

Example Usage:

```
```python
print(is_valid_email("example@example.com")) # True
print(is_valid_email("user.name+tag+sorting@example.com")) # True
print(is_valid_email("user@sub.example.com")) # True
print(is_valid_email("invalid-email.com")) # False
print(is_valid_email("user@.com")) # False
```
```

This implementation handles most basic email validation cases, but for more complex scenarios, additional checks might be needed.

Response time: 5.57 seconds

Llama-3.1 Response:

```
```python
import re

def is_valid_email(email):
 """
 Validates a basic email address.
 Args:
 email (str): The email address to be validated.
 Returns:
 bool: True if the email is valid, False otherwise.
 """
 # Regular expression pattern to match basic email addresses
 pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
 # Use the re.match function to check if the email matches the pattern
 # re.match only matches from the beginning of the string
 if re.match(pattern, email):
 return True
 else:
 return False
Example use cases:
print(is_valid_email("test@example.com")) # Expected output: True
print(is_valid_email("invalid_email")) # Expected output: False
print(is_valid_email("test@example")) # Expected output: False
print(is_valid_email("test@example.com.au")) # Expected output: True
This function uses a regular expression to match basic email addresses. The pattern used in this function checks for the following:
- ^ asserts the start of the line.
- [a-zA-Z0-9._%+-]+ matches one or more alphanumeric characters, dots, underscores, percent signs, plus signs, or hyphens.
- @ matches the @ symbol.
```

- `[a-zA-Z0-9.-]+` matches one or more alphanumeric characters, dots, or hyphens.
- `\. ` matches a dot (.) symbol.
- `[a-zA-Z]{2,}` matches the domain extension (it must be at least 2 characters long).
- `\$` asserts the end of the line.

Note that this is a basic validation and may not cover all possible valid email formats according to the official specification (RFC 5322). For more advanced validation, consider using a dedicated library or a more complex regular expression.

Response time: 13.27 seconds

### Few-shot Prompt:

Complete the function using regex to validate basic email addresses:

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

Example 1:

```
def is_valid_email(email):
 import re
 pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
 return bool(re.match(pattern, email))
```

Example 2:

```
def is_valid_email(email):
 import re
 email_pattern = r'^[\w.-]+@[[\w.-]+\.[a-zA-Z]{2,}$'
 if re.match(email_pattern, email):
 return True
 return False
```

### GPT-4o Response:

Here's the completed `is\_valid\_email` function using regex:

```
```python
def is_valid_email(email):
    import re
    # Define the regex pattern for basic email validation
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    # Check if the email matches the pattern
    return bool(re.match(pattern, email))
```
```

### Explanation of the Regex Pattern:

- `^`: Asserts the start of the string.
- `[a-zA-Z0-9.\_%+-]+`: Matches one or more alphanumeric characters (`a-z`, `A-Z`, `0-9`) or special characters like `.`, `\_`, `%`, `+`, `-`.
- `@`: Matches the `@` symbol (required in all valid email addresses).
- `[a-zA-Z0-9.-]+`: Matches the domain name part, which can include alphanumeric characters, `.` and `-`.
- `\.`: Matches a literal dot (.), which separates the domain name and the top-level domain (TLD).

```
- `[a-zA-Z]{2,}`: Matches the TLD, which must be at least two alphabetic characters long (e.g., `.com`,
`.org`).
- `$`: Asserts the end of the string.
Example Usage:
```python
print(is_valid_email("example@example.com")) # True
print(is_valid_email("user.name+tag@domain.co.uk")) # True
print(is_valid_email("invalid-email@com")) # False
print(is_valid_email("missingatsign.com")) # False
print(is_valid_email("user@.invalid")) # False
```
```

This solution ensures that the email address follows a basic structure, though it doesn't cover all edge cases or obscure valid email formats (e.g., very long domain names or special characters in domain names). For more robust email validation, other libraries or tools (like Python's `email` module) might be more appropriate.

Response time: 6.33 seconds

Llama-3.1 Response:

```
Here's the completed function using regex to validate basic email addresses:
'''
python
import re
def is_valid_email(email):
 """
 Validate a basic email address using regex.
 Args:
 email (str): The email address to validate.
 Returns:
 bool: True if the email address is valid, False otherwise.
 """
 email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
 return bool(re.match(email_pattern, email))
Example usage:
print(is_valid_email("test@example.com")) # Returns: True
print(is_valid_email("invalid_email")) # Returns: False
'''
```

Explanation:

- \* The regex pattern `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$` breaks down as follows:
  - \* `^` asserts the start of the string.`
  - \* ``[a-zA-Z0-9._%+-]+` matches one or more alphanumeric characters, dots, underscores, percent signs, plus signs, or hyphens (valid characters for the local part of an email address).`
  - \* ``@` matches the @ symbol, separating the local part from the domain.`
  - \* ``[a-zA-Z0-9.-]+` matches one or more alphanumeric characters, dots, or hyphens (valid characters for the domain).`
  - \* ``\.` matches a dot (escaped with a backslash because dot has a special meaning in regex).`
  - \* ``[a-zA-Z]{2,}` matches the domain extension (it must be at least 2 characters long).`
  - \* ``$` asserts the end of the string.`

\* The `re.match` function attempts to match the regex pattern against the input email address. If a match is found, it returns a match object; otherwise, it returns `None`.

\* The `bool` function is used to convert the match object to a boolean value (`True` if a match is found, `False` otherwise).

Note that this is a basic email validation and does not cover all possible valid email formats according to the official specification (RFC 5322). For more complex email validation, consider using a dedicated library or a more comprehensive regex pattern.

Response time: 16.77 seconds

**Analysis:** Both models produced similar regex-based email validators. GPT-4o stood out with detailed explanations and examples (e.g., `is_valid_email("example@example.com")`), while Llama's responses were functional but less explanatory, although it did include function header comments. GPT-4o was faster (5.57s vs. 13.27s in zero-shot). Few-Shot prompt type had minimal impact due to the task's simplicity.

**Comparison:** Prefer GPT-4o for speed, clarity and guidance.

## Task 5: Generating Flask APIs (Python)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

```
Here's how you can complete the `greet` endpoint to return a JSON greeting:
...
python
from flask import Flask, jsonify
app = Flask(__name__)

@app.route('/greet/<username>')
def greet(username):
 # Create a JSON response with a greeting message
 return jsonify({"message": f"Hello, {username}!"})
if __name__ == "__main__":
 app.run(debug=True)
...

Explanation:
1. jsonify: This function is used to create a properly formatted JSON response.
```

2. **\*\*Dynamic Route\*\***: ``<username>`` is a dynamic component in the route that allows users to pass their username as part of the URL.
3. **\*\*Response\*\***: The ``jsonify`` function takes a dictionary (or other serializable data) and converts it to a JSON object that is returned to the client.

### Example Usage:

- Start the Flask app by running the script: ``python app.py``.
- Open a browser or use ``curl``/Postman and visit the endpoint, e.g., ``http://127.0.0.1:5000/greet/Alice``.
- You will get the following JSON response:

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

Response time: 15.51 seconds

Llama-3.1 Response:

Here's how you can implement the endpoint to return a JSON greeting:

```
'''
python
from flask import Flask, jsonify
app = Flask(__name__)
@app.route('/greet/<username>')
def greet(username):
 """Returns a JSON greeting for the given username"""
 greeting = {"message": f"Hello, {username}!"}
 return jsonify(greeting)
Example usage:
GET /greet/John
Response: {"message": "Hello, John!"}
'''
```

In this code:

1. We define a dictionary ``greeting`` with a single key-value pair, where the key is ``"message"`` and the value is a personalized greeting for the given ``username``.
  2. We use the ``jsonify`` function from Flask to convert the ``greeting`` dictionary into a JSON response.
- You can test this endpoint by sending a GET request to ``/greet/<username>``, replacing ``<username>`` with the desired username. The response should be a JSON object with a personalized greeting message.

Response time: 21.76 seconds

Few-shot Prompt:

Create a ``/greet/<username>`' endpoint that returns a JSON greeting:

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



```
pass
```

Example 1:

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

Example 2:

```
@app.route('/greet/<username>')
def greet(username):
 response = {
 "greeting": f"Welcome, {username}!",
 "timestamp": datetime.now().isoformat()
 }
 return jsonify(response)
```

GPT-4o Response:

Below is the complete Flask code:

```
```python
from flask import Flask, jsonify
from datetime import datetime
app = Flask(__name__)
@app.route('/greet/<username>')
def greet(username):
    # Example of a JSON response with a greeting and a timestamp
    response = {
        "greeting": f"Hello, {username}!",
        "timestamp": datetime.now().isoformat() # Adds the current timestamp
    }
    return jsonify(response)
if __name__ == '__main__':
    app.run(debug=True)
```
```

### Explanation:

1. **Simple Greeting**: The ``greeting`` key contains a personalized message for the provided ``username``.
2. **Timestamp (optional)**: The ``timestamp`` key includes the current date and time in ISO 8601 format (``datetime.now().isoformat()``).
3. **jsonify**: Used to create a valid JSON response from the dictionary.

### Example Outputs:

```
1. Request: `GET /greet/Alice`
Response:
```json
{
    "greeting": "Hello, Alice!",
    "timestamp": "2023-10-05T12:34:56.789123"
}
```

```

...
2. **Request**: `GET /greet/Bob`
**Response**:
```json
{
 "greeting": "Hello, Bob!",
 "timestamp": "2023-10-05T12:35:10.123456"
}
...

```

Feel free to modify the structure of the JSON response or add additional fields as needed!

Response time: 6.58 seconds

Llama-3.1 Response:

Here's a simple implementation of the endpoint using Flask:

```

...
python
from flask import Flask, jsonify
from datetime import datetime
app = Flask(__name__)
@app.route('/greet/<username>')
def greet(username):
 """Return a JSON greeting."""
 response = {
 "message": f"Hello, {username}!",
 "timestamp": datetime.now().isoformat()
 }
 return jsonify(response)
if __name__ == '__main__':
 app.run(debug=True)
...

```

### Explanation:

- \* We define a route `/greet/<username>` that accepts a `username` parameter.
- \* The `greet` function constructs a JSON response with a greeting message and the current timestamp.
- \* We use the `jsonify` function provided by Flask to return the response as a JSON object.

### Example Usage:

- \* Send a GET request to `http://localhost:5000/greet/John` to receive a JSON greeting.

### Response:

```

```json
{
  "message": "Hello, John!",
  "timestamp": "2023-12-01T12:00:00.000000"
}
...

```

Running the Application:

- * Save the code in a file (e.g., `app.py`).
- * Install Flask using pip: `pip install flask`

```
* Run the application using `python app.py`
* Access the endpoint in your web browser or using a tool like curl: `curl http://localhost:5000/greet/John`
```

Response time: 10.70 seconds

Analysis: With Zero-Shot both were able to generate the greet endpoint. GPT-4o included the python code, and explanation, and usage examples. Llama included explanations in sentence format and a vague way to test the greeting. Llama also included a redundant variable “greeting”. Few-shot prompting improved and extended outputs. GPT-4o generated a concise, complete greet endpoint with a timestamp, while Llama provided additional information on how to run the application. I found GPT-4o’s response excellent, but I really liked Llama’s instructions on running the application.

Comparison: Use GPT-4o for efficient code or Llama for learning-focused outputs.

Task 6: SQL Schema Design (SQL)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

Below is the SQL schema for a review app with `users`, `books`, and `reviews` tables. The schema includes appropriate keys, constraints, and relationships to ensure data integrity.

```
```sql
-- USERS table
CREATE TABLE users (
 id SERIAL PRIMARY KEY, -- Primary key, auto-incremented
 name VARCHAR(255) NOT NULL -- User name, required field
);
-- BOOKS table
CREATE TABLE books (
 id SERIAL PRIMARY KEY, -- Primary key, auto-incremented
 title VARCHAR(255) NOT NULL -- Book title, required field
);
-- REVIEWS table
CREATE TABLE reviews (
 id SERIAL PRIMARY KEY, -- Primary key, auto-incremented
 user_id INT NOT NULL, -- Foreign key referencing the users table
 book_id INT NOT NULL, -- Foreign key referencing the books table
 rating INT NOT NULL CHECK (rating >= 1 AND rating <= 5), -- Rating between 1 and 5
 review_text TEXT, -- Optional text for the review
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- Review creation timestamp
 -- Constraints
```

```
CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users (id) ON DELETE CASCADE,
CONSTRAINT fk_book FOREIGN KEY (book_id) REFERENCES books (id) ON DELETE CASCADE,
CONSTRAINT unique_user_book UNIQUE (user_id, book_id) -- Ensures a user cannot review the
same book twice
```

```
);
...
```

#### ### Explanation of the Schema Design

##### 1. **Users Table**:

- Contains a unique `id` for each user and a `name` field.
- The `id` is the primary key and is auto-incremented (`SERIAL`).

##### 2. **Books Table**:

- Contains a unique `id` for each book and a `title` field.
- The `id` is the primary key and is auto-incremented (`SERIAL`).

##### 3. **Reviews Table**:

- Contains a unique `id` for each review.
- Includes `user\_id` and `book\_id` as foreign keys referencing the `users` and `books` tables, respectively.
- Includes a `rating` field restricted to values between 1 and 5 using a `CHECK` constraint.
- Includes an optional `review\_text` field for additional comments.
- Includes a `created\_at` timestamp to track when the review was created.
- Enforces a `UNIQUE` constraint (`unique\_user\_book`) to ensure that a user cannot review the same book more than once.
- Uses `ON DELETE CASCADE` for foreign key constraints to ensure that if a user or book is deleted, associated reviews are also deleted automatically.

#### ### Example Usage

##### 1. Adding a user:

```
```sql  
INSERT INTO users (name) VALUES ('John Doe');  
```
```

##### 2. Adding a book:

```
```sql  
INSERT INTO books (title) VALUES ('The Great Gatsby');  
```
```

##### 3. Adding a review:

```
```sql  
INSERT INTO reviews (user_id, book_id, rating, review_text)  
VALUES (1, 1, 5, 'Amazing book!');  
```
```

Response time: 9.97 seconds

Llama-3.1 Response:

```
```sql  
-- Create the schema for the review app  
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,
  book_id INTEGER NOT NULL,
  rating SMALLINT NOT NULL CHECK (rating BETWEEN 1 AND 5),
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
  FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE,
  UNIQUE (user_id, book_id) -- Ensure each user can only review a book once
);
...
```

In this schema:

1. The `users` table stores information about the users, with a unique `id` and a `name`.
2. The `books` table stores information about the books, with a unique `id` and a `title`.
3. The `reviews` table stores the reviews, with foreign keys referencing the `users` and `books` tables. This ensures data consistency and allows for efficient querying.
4. The `rating` field in the `reviews` table is constrained to be between 1 and 5, inclusive.
5. The `UNIQUE` constraint in the `reviews` table ensures that each user can only review a book once.
6. The `ON DELETE CASCADE` clause in the foreign key constraints ensures that when a user or book is deleted, all associated reviews are also deleted. This maintains data consistency and prevents orphaned reviews.

Response time: 11.34 seconds

Few-shot Prompt:

```
Write the schema for a review app with users, books, and reviews:
-- TODO: Design schema with appropriate keys and constraints
-- Tables: users(id, name), books(id, title), reviews(id, user_id, book_id, rating)
Example 1:
CREATE TABLE users (
  id INTEGER PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);

CREATE TABLE books (
  id INTEGER PRIMARY KEY,
  title VARCHAR(200) NOT NULL
);
CREATE TABLE reviews (
  id INTEGER PRIMARY KEY,
  user_id INTEGER NOT NULL,
  book_id INTEGER NOT NULL,
  rating INTEGER CHECK(rating BETWEEN 1 AND 5),
  FOREIGN KEY (user_id) REFERENCES users(id),
  FOREIGN KEY (book_id) REFERENCES books(id)
);
```

Example 2:

```
CREATE TABLE users (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  name VARCHAR(100) NOT NULL,  
  email VARCHAR(100) UNIQUE  
);  
CREATE TABLE books (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  title VARCHAR(200) NOT NULL,  
  author VARCHAR(100),  
  published_year INTEGER  
);  
CREATE TABLE reviews (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  user_id INTEGER NOT NULL,  
  book_id INTEGER NOT NULL,  
  rating INTEGER NOT NULL CHECK(rating BETWEEN 1 AND 5),  
  review_text TEXT,  
  created_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  
);
```

GPT-4o Response:

Here's a full schema design for a review application with `users`, `books`, and `reviews` tables. This schema includes appropriate keys, constraints, and additional attributes to make the system more functional and robust:

```
```sql  
-- Users Table
CREATE TABLE users (
 id INTEGER PRIMARY KEY AUTOINCREMENT, -- Auto-incrementing primary key
 name VARCHAR(100) NOT NULL, -- User's name, cannot be null
 email VARCHAR(100) UNIQUE NOT NULL, -- Unique and non-null email for each user
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Timestamp for when the user is created
);
-- Books Table
CREATE TABLE books (
 id INTEGER PRIMARY KEY AUTOINCREMENT, -- Auto-incrementing primary key
 title VARCHAR(200) NOT NULL, -- Book title, cannot be null
 author VARCHAR(100), -- Book's author
 published_year INTEGER CHECK (published_year >= 0), -- Optional publication year (checked for
non-negative values)
 genre VARCHAR(50), -- Optional genre of the book
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Timestamp for when the book is added
);
-- Reviews Table
CREATE TABLE reviews (
 id INTEGER PRIMARY KEY AUTOINCREMENT, -- Auto-incrementing primary key
 user_id INTEGER NOT NULL, -- User who wrote the review
 book_id INTEGER NOT NULL, -- Book being reviewed
 rating INTEGER NOT NULL CHECK(rating BETWEEN 1 AND 5),
 review_text TEXT, -- Review content
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Timestamp for when the review is added
);
```

```

id INTEGER PRIMARY KEY AUTOINCREMENT, -- Auto-incrementing primary key
user_id INTEGER NOT NULL, -- Foreign key to the users table
book_id INTEGER NOT NULL, -- Foreign key to the books table
rating INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5), -- Rating is required, must be
between 1 and 5
review_text TEXT, -- Optional text for the review
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- Timestamp for when the review is
created
FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE, -- Cascading delete for
users
FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE -- Cascading delete for
books
);
-- Indexes for faster queries and lookups
CREATE INDEX idx_reviews_user_id ON reviews(user_id); -- Index on reviews.user_id
CREATE INDEX idx_reviews_book_id ON reviews(book_id); -- Index on reviews.book_id
CREATE INDEX idx_books_author ON books(author); -- Index on books.author
CREATE INDEX idx_users_email ON users(email); -- Index on users.email
...

```

#### ### Explanation of Design Choices:

##### 1. **users` Table**:

- `id`: Primary key with auto-increment.
- `name`: Non-null attribute for the user's name.
- `email`: Unique and non-null to ensure no duplicate accounts.
- `created\_at`: Tracks when the user was registered.

##### 2. **books` Table**:

- `id`: Primary key with auto-increment.
- `title`: Non-null attribute for the book's title.
- `author`: Optional attribute for the book's author.
- `published\_year`: Optional attribute with a constraint to ensure non-negative values.
- `genre`: Optional attribute to classify the book.
- `created\_at`: Tracks when the book was added to the database.

##### 3. **reviews` Table**:

- `id`: Primary key with auto-increment.
- `user\_id`: Foreign key referencing `users(id)`, with cascading delete to remove reviews if a user is deleted.
- `book\_id`: Foreign key referencing `books(id)`, with cascading delete to remove reviews if a book is deleted.
- `rating`: Non-null attribute with a constraint to ensure ratings are between 1 and 5.
- `review\_text`: Optional field for users to add review details.
- `created\_at`: Tracks when the review was added.

##### 4. **Indexes**:

- Added indexes on `reviews.user\_id`, `reviews.book\_id`, `books.author`, and `users.email` to optimize queries related to these fields.

This schema provides a robust foundation for a review app with proper relationships, constraints, and features like timestamps and cascading deletes for better data integrity.

Response time: 11.43 seconds

Llama-3.1 Response:

Based on the provided examples, I'll create a more comprehensive schema for the review app. Here's the updated schema:

```
```sql
-- Create the users table
CREATE TABLE users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create the books table
CREATE TABLE books (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title VARCHAR(200) NOT NULL,
  author VARCHAR(100),
  published_year INTEGER,
  isbn VARCHAR(20) UNIQUE,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create the reviews table
CREATE TABLE reviews (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER NOT NULL,
  book_id INTEGER NOT NULL,
  rating INTEGER NOT NULL CHECK(rating BETWEEN 1 AND 5),
  review_text TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
  FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE
);

-- Create an index on user_id and book_id for faster lookup
CREATE INDEX idx_reviews_user_id ON reviews (user_id);
CREATE INDEX idx_reviews_book_id ON reviews (book_id);

-- Create a trigger to update the updated_at timestamp
CREATE TRIGGER trg_users_updated_at
AFTER UPDATE ON users
FOR EACH ROW
BEGIN
  UPDATE users SET updated_at = CURRENT_TIMESTAMP WHERE id = NEW.id;
END;

CREATE TRIGGER trg_books_updated_at
AFTER UPDATE ON books
```



```

FOR EACH ROW
BEGIN
  UPDATE books SET updated_at = CURRENT_TIMESTAMP WHERE id = NEW.id;
END;
CREATE TRIGGER trg_reviews_updated_at
AFTER UPDATE ON reviews
FOR EACH ROW
BEGIN
  UPDATE reviews SET updated_at = CURRENT_TIMESTAMP WHERE id = NEW.id;
END;
'''

```

This schema includes the following enhancements:

- * Added `email` and `password` fields to the `users` table.
- * Added `author`, `published_year`, and `isbn` fields to the `books` table.
- * Added `review_text` field to the `reviews` table.
- * Added `created_at` and `updated_at` timestamps to all tables.
- * Created indexes on `user_id` and `book_id` in the `reviews` table for faster lookup.
- * Created triggers to update the `updated_at` timestamp when a record is updated in any of the tables.

These enhancements provide a more comprehensive schema for the review app, allowing for more detailed user information, book metadata, and review text, as well as improved performance and data consistency.

Response time: 19.44 seconds

Analysis: for Zero-Prompt both created SQL schemas for a database of book reviews. GPT-4o included a timestamp detail, Llama reverse cascade delete just in case a book was deleted the associated reviews would be removed.

for Few-Shot, GPT-4o created a full fledged schema including user timestamps, author data, publishing data, genre, optional review text, cascading delete, etc.. It further goes on to explain its design choices. Llama excelled with detailed schemas including user email, passwords, updated dates, and isbn numbering. Furthermore it suggested Triggers for updating timestamps

Comparison: While GPT-4o offered a robust, straightforward, fast solution. Llama created a more comprehensive design with additional features.

Task 7: Null Dereference Detection (Java)

Prompting Strategy 1: Zero-shot Prompt:

```

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

```

GPT-4o Response:

Yes, there is a **null dereference risk** in the given code. If the input `String s` is `null`, calling `s.length()` will throw a **NullPointerException**.

Explanation:

The method `s.length()` is invoked without checking if `s` is `null`. If the caller passes `null` as the argument to the `getLength` method, the program will crash with a `NullPointerException`.

How to Fix It:

You can add a `null` check to handle this case gracefully. For example:

Option 1: Return a Default Value (e.g., 0)

```
```java
public int getLength(String s) {
 if (s == null) {
 return 0; // Default value for null input
 }
 return s.length();
}
...
```
```

This approach is useful if it makes sense for your application to treat `null` as a string of length `0`.

Option 2: Throw an Exception

If `null` is an invalid input and should not be allowed, throw an exception:

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

This approach forces the caller to handle the `null` case explicitly.

Option 3: Use `Optional` (Java 8+)

You can use `Optional` to make `null` handling explicit:

```
```java
public int getLength(Optional<String> optionalString) {
 return optionalString
 .map(String::length)
 .orElse(0); // Default to 0 if the Optional is empty
}
...
```
```

The caller would then pass an `Optional<String>` instead of a `String`.

Summary:

The original method has a **null dereference risk** and should be updated to handle `null` inputs in a way that aligns with the application's requirements.

Response time: 7.93 seconds

Llama-3.1 Response:

****Null Dereference Risk:****

The given code snippet has a null dereference risk. If the input string `s` is null, calling `s.length()` will result in a `NullPointerException`. This is because you're attempting to access the `length()` method on a null object.

****Mitigation Strategies:****

To avoid this risk, you can implement the following strategies:

1. Null Check

Perform a null check before calling the `length()` method:

```
```java
public int getLength(String s) {
 if (s == null) {
 throw new NullPointerException("Input string is null");
 }
 return s.length();
}
...
```
```

2. Optional Class (Java 8+)

Use the `Optional` class to handle null values:

```
```java
public int getLength(String s) {
 return Optional.ofNullable(s)
 .map(String::length)
 .orElseThrow(() -> new NullPointerException("Input string is null"));
}
...
```
```

3. Null Safety with Objects (Java 7-)

Use the `Objects.requireNonNull()` method to check for null values:

```
```java
public int getLength(String s) {
 Objects.requireNonNull(s, "Input string is null");
 return s.length();
}
...
```
```

4. Default Value

Return a default value when the input string is null:

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

Choose the strategy that best fits your use case and error handling requirements.

Response time: 10.26 seconds

Few-shot Prompt:

Identify any null dereference risk:

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

```
}
```

Example 1:

There is a null dereference risk in this function. If the parameter 's' is null, calling s.length() will throw a NullPointerException. The function should check if 's' is null before attempting to access its length.

Example 2:

This method has a null dereference vulnerability. When 's' is null, calling length() on it will cause a NullPointerException. To fix this, add a null check before accessing the length property, or use Objects.requireNonNull() to validate the input.

GPT-4o Response:

Both examples correctly identify the null dereference risk in the provided function. However, they provide slightly different explanations and solutions.

Analysis of the Code:

The method `getLength` attempts to call the `length()` method on the parameter `s`. If `s` is null, this will result in a `NullPointerException` because the method is trying to dereference a null reference.

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

Example 1:

This example provides a correct and concise identification of the issue:

- It explicitly mentions that if the parameter `s` is null, a `NullPointerException` will be thrown.
- It suggests adding a null check to prevent this issue.

Example 2:

This example also correctly identifies the issue and provides more specific guidance for fixing it:

- It explains the null dereference vulnerability in a similar way.
- It offers two solutions for fixing the problem:
 1. Add a null check before accessing the `length` property.
 2. Use `Objects.requireNonNull()` to validate the input.

Suggested Fix:

Here's how you can address the null dereference risk:

Null Check:

You can add an explicit null check to handle the situation gracefully:

```
```java
public int getLength(String s) {
 if (s == null) {
 return 0; // or throw an IllegalArgumentException
 }
 return s.length();
}
...
```
```

Using `Objects.requireNonNull()`:

You can use `Objects.requireNonNull()` from the `java.util` package to enforce that `s` is not null:

```

```java
import java.util.Objects;

public int getLength(String s) {
 Objects.requireNonNull(s, "Input string cannot be null");
 return s.length();
}
...

```

If `s` is null, this will throw a `NullPointerException` with the specified message, making the error easier to debug.

#### ### Conclusion:

Both Example 1 and Example 2 accurately identify the null dereference risk. However, Example 2 provides a slightly more comprehensive explanation by also suggesting the use of `Objects.requireNonNull()` as an alternative solution. Depending on your use case and coding style, either approach can be used to mitigate the risk.

Response time: 42.17 seconds

#### Llama-3.1 Response:

The provided code snippet is vulnerable to a null dereference risk:

```

```java
public int getLength(String s) {
    return s.length(); // Risks NullPointerException if 's' is null
}
...

```

Problem Explanation

When the `getLength` method is called with a `null` argument, it attempts to access the `length` property of the `null` object, resulting in a `NullPointerException`. This can lead to unexpected crashes, errors, or security vulnerabilities.

Mitigation Strategies

To eliminate the null dereference risk, you can use one of the following approaches:

1. **Null Check**

Perform an explicit null check before accessing the `length` property:

```

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

```

##### ### 2. \*\*Objects.requireNonNull()

Use the `Objects.requireNonNull()` method to validate the input and throw a `NullPointerException` with a meaningful error message:

```

```java
import java.util.Objects;

public int getLength(String s) {
    Objects.requireNonNull(s, "Input string cannot be null");
}
...

```

```
    return s.length();
}
```

3. **Optional Class (Java 8 and later)**

Use the `Optional` class to handle the possibility of a `null` input:

```
```java
import java.util.Optional;

public int getLength(String s) {
 return Optional.ofNullable(s)
 .map(String::length)
 .orElseThrow(() -> new NullPointerException("Input string cannot be null"));
}
```
```

By incorporating one of these null dereference mitigation strategies, you can ensure that your code is more robust, secure, and less prone to unexpected errors.

Response time: 11.88 seconds

Analysis: Both models spotted the null dereference risk. GPT-4o provided a clear explanation with multiple fixes (e.g., default values, exceptions), while Llama included an Optional class to handle the situation. Prompting style had little effect on accuracy but did add approx 30 seconds of processing time for GPT-4o and additional wordy explanations undesired.

Comparison: This a tossup as to which to use. They are both accurate and complete but I would avoid Few-Shot.

Task 8: CSV Parser Variants (Python)

Prompting Strategy 1: Zero-shot Prompt:

Improve the parser to support quoted fields:

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

GPT-4o Response:

To handle quoted fields in a CSV line properly, you can use Python's built-in `csv` module, which is specifically designed to handle various edge cases in CSV parsing, including quoted fields. Here's the improved `parse_csv_line` function:

```
```python
import csv
from io import StringIO
def parse_csv_line(line):
 # Use the csv.reader to handle quoted fields
 csv_reader = csv.reader(StringIO(line), quotechar='"', delimiter=',', skipinitialspace=True)
 return next(csv_reader) # Return the parsed line as a list
```
```

```
# Example usage:
line = '1,"hello, world",3.14,"a ""quoted"" string"'
parsed_line = parse_csv_line(line)
print(parsed_line)
'''

#### Explanation:
1. csv.reader: The csv.reader is used to parse the CSV line. It handles quoted fields, escaped quotes, and other CSV-specific nuances.
2. StringIO: Since csv.reader expects a file-like object, we wrap the line string in a StringIO object to simulate a file.
3. quotechar='\"': Specifies that double quotes (") are used to quote fields.
4. delimiter=',': Specifies that the fields are separated by commas.
5. skipinitialspace=True: Ensures that spaces following a delimiter are ignored, which is common in CSV files.

#### Example Input and Output:
Input:
```python
line = '1,"hello, world",3.14,"a ""quoted"" string"'
'''

Output:
```python
['1', 'hello, world', '3.14', 'a "quoted" string']
'''

This implementation ensures that quoted fields are parsed correctly, including handling commas inside quotes and escaped quotes.
```

Response time: 5.45 seconds

Llama-3.1 Response:

Here's a Python function that can parse a CSV line with quoted fields. This function uses a simple state machine to track whether we are inside a quoted field or not.

```
```python
def parse_csv_line(line):
 """
 Parse a CSV line into a list of fields, handling quoted fields.
 Args:
 line (str): The CSV line to parse.
 Returns:
 list[str]: A list of fields in the CSV line.
 """
 fields = []
 field = ""
 in_quote = False
 for char in line:
 if char == '"':
 in_quote = not in_quote
 elif char == ',' and not in_quote:
 fields.append(field)
 field = ""
 fields.append(field)
 return fields
```
```

```

        fields.append(field)
        field = ""
    else:
        field += char
    fields.append(field)
    # Remove leading/trailing quotes from quoted fields
    fields = [field.strip('"') for field in fields]
    return fields
'''
#### Example Use Cases
```python
Test with a simple CSV line
print(parse_csv_line("hello,world,python")) # Output: ['hello', 'world', 'python']
Test with a CSV line containing quoted fields
print(parse_csv_line('"hello, world",python,"a,b,c"')) # Output: ['hello, world', 'python', 'a,b,c']
'''

Using the `csv` Module
For more complex CSV parsing tasks, consider using Python's built-in `csv` module, which provides a
more robust and flexible way to parse CSV files.
```python
import csv

def parse_csv_line(line):
    return next(csv.reader([line]))
# Test with a CSV line containing quoted fields
print(parse_csv_line('"hello, world",python,"a,b,c"')) # Output: ['hello, world', 'python', 'a,b,c']
'''

```

Response time: 13.42 seconds

Few-shot Prompt:

```

Improve the parser to support quoted fields:
def parse_csv_line(line):
    return line.split(',') # Incomplete: doesn't handle quoted fields

Example 1:
def parse_csv_line(line):
    import csv
    return next(csv.reader([line]))

Example 2:
def parse_csv_line(line):
    import re
    pattern = r'(?=(?:[^\"]*"|"[^"]*"|'')*$)'
    fields = re.split(pattern, line)
    return [field.strip('"') for field in fields]

```

GPT-4o Response:

To enhance the `parse_csv_line` function to support quoted fields, we can follow both approaches demonstrated in the examples. Each has its own advantages. Here's an explanation and improvement of the two approaches:

```
---
### **Example 1: Using the `csv` module (preferred approach)**
The `csv` module in Python is specifically designed to handle CSV parsing, including quoted fields, escape sequences, and other edge cases. Using this module ensures robust parsing without reinventing the wheel.
Improved version:
```python
def parse_csv_line(line):
 import csv
 # Use csv.reader to handle quoted fields and delimiters properly
 return next(csv.reader([line]))
```

**Advantages:**
- Handles edge cases like escaped quotes (`` inside quotes).
- Built-in handling of different delimiters (if needed, you can customize the `csv.reader` with parameters like `delimiter` or `quotechar`).
- Simple and reliable.
**Example usage:**
```python
line = '1,"Hello, World!",3.14,"""Quoted"" text"'
print(parse_csv_line(line))
Output: ['1', 'Hello, World!', '3.14', '"Quoted" text']
```
---
### **Example 2: Using Regular Expressions (manual approach)**
If for some reason you cannot use the `csv` module, you can use regular expressions to split the CSV line while respecting quoted fields. Here's a revised and improved version of the regex-based approach:
```python
def parse_csv_line(line):
 import re
 # Regex pattern to split by commas, but ignore commas inside quotes
 pattern = r',(?=(?:[^\"]*"[""]*)*[^\"]*$)'
 fields = re.split(pattern, line)
 # Strip surrounding quotes and handle escaped quotes
 return [re.sub(r'^"|"$', "", field).replace('""', '"') for field in fields]
```

**Explanation:**
1. r',(?=(?:[^\"]*"[""]*)*[^\"]*$)':
    - Matches commas `,` that are not inside quoted strings.
    - Uses lookahead to ensure the number of quotes before and after the comma is balanced.
2. re.sub(r'^"|"$', "", field):
    - Strips leading and trailing quotes from each field.
3. .replace('""', '"'):
    - Replaces escaped quotes (``) with a single quote (").
**Example usage:**

```

```

python
line = '1,"Hello, World!",3.14,""Quoted"" text"'
print(parse_csv_line(line))
# Output: ['1', 'Hello, World!', '3.14', '"Quoted" text']
'''
---
### **Comparison of Approaches**
**Criteria**	**`csv` module**	**Regex-based approach**
**Ease of use**	Very easy	Moderate complexity
**Performance**	Optimized	Slower for complex cases
**Edge case handling**	Comprehensive	Requires manual handling
**Maintainability**	High	Lower
---
### **Recommendation**
Use the `csv` module approach (Example 1) unless you have specific constraints that prevent its usage. It is more robust, reliable, and easier to maintain. However, if you need to customize your parsing logic or cannot use external libraries, the regex-based approach (Example 2) is a viable alternative.
Response time: 10.58 seconds

```

Llama-3.1 Response:

```

**Improved CSV Parser with Quoted Field Support**
To enhance the CSV parser and handle quoted fields, we can implement a more robust solution using the built-in `csv` module or regular expressions.
### Method 1: Using the `csv` Module
python
import csv
def parse_csv_line(line):
    """
    Parse a CSV line with quoted field support using the csv module.
    Args:
        line (str): A single CSV line.
    Returns:
        list: A list of parsed fields.
    """
    return next(csv.reader([line]))
'''
**Example Use Case:**
python
line = 'Name,"Age, Address",Country'
parsed_fields = parse_csv_line(line)
print(parsed_fields) # Output: ['Name', 'Age, Address', 'Country']
'''
### Method 2: Using Regular Expressions
python
import re

```

```
def parse_csv_line(line):
    """
    Parse a CSV line with quoted field support using regular expressions.
    Args:
        line (str): A single CSV line.
    Returns:
        list: A list of parsed fields.
    """
    pattern = r'(?=(?:[^\"]*"|"[^"]*"|'')[^"]*$)'
    fields = re.split(pattern, line)
    return [field.strip('"') for field in fields]
'''
**Example Use Case:**
```python
line = 'Name,"Age, Address",Country'
parsed_fields = parse_csv_line(line)
print(parsed_fields) # Output: ['Name', 'Age, Address', 'Country']
```
**Comparison of Methods:**
While both methods achieve the desired result, the `csv` module approach is generally more efficient and robust. The `csv` module is designed specifically for parsing CSV files and handles various edge cases, such as escaped quotes and newline characters. The regular expression method, on the other hand, may be more prone to errors and performance issues for large datasets.
**Recommendation:**
Unless there's a specific requirement to use regular expressions, the `csv` module approach is recommended for its simplicity, efficiency, and reliability.
Response time: 5.97 seconds
```

Analysis: in Zero-Prompt GPT-4o went straight to using the built in Python csv parser, while Llama rewrote the code to explicitly handle the quoted fields. At the end Llama suggests for complex parsing to use the built in parser. This might in fact be what we wanted. With Few-Shot we ask for the built in csv parser explicitly, but the second example asks for use of regular expressions. Both offer a two function solution to this, one using the built in parser and the other based on regular expressions. GPT-4o points out that using the built in parser offers greater robustness, but allows that the user might want to write their own custom version. It further offers a comparison based on certain criteria. Very complete and very detailed. Llama's solution is less robust and offers less to be learned.

Comparison: Use GPT-4o for standard solutions and detailed explanations.

Task 9: Data Class to API Conversion (Kotlin)

Prompting Strategy 1: Zero-shot Prompt:

```
Convert the data class to a REST API using Ktor:
data class Product(val id: Int, val name: String, val price: Double)
```

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

GPT-4o Response:

To convert the `Product` data class into a REST API using Ktor, you can define GET and POST endpoints to retrieve and add products. Here's an example implementation:

```
```kotlin
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.server.engine.*
import io.ktor.server.netty.*
import kotlinx.serialization.*
import kotlinx.serialization.json.Json
// Define the Product data class
@Serializable
data class Product(val id: Int, val name: String, val price: Double)

// In-memory product storage
val products = mutableListOf<Product>()
fun main() {
 embeddedServer(Netty, port = 8080) {
 module()
 }.start(wait = true)
}
fun Application.module() {
 install(io.ktor.features.ContentNegotiation) {
 io.ktor.serialization.kotlinx.json()
 }
 routing {
 // GET endpoint: Retrieve all products
 get("/products") {
 call.respond(products)
 }
 // POST endpoint: Add a new product
 post("/products") {
 val newProduct = call.receive<Product>()
 products.add(newProduct)
 call.respond(HttpStatusCode.Created, newProduct)
 }
 }
}
```
```

Explanation:

1. **Data Class**:

- The `Product` class is annotated with `@Serializable` to allow JSON serialization/deserialization using Kotlinx Serialization.

2. ****In-Memory Storage****:
 - A mutable list (`products`) is used to store products. In a real-world application, this would connect to a database.
3. ****Install Plugins****:
 - The `ContentNegotiation` feature with `kotlinx.json` is installed to handle JSON serialization.
4. ****GET Endpoint****:
 - The `/products` route retrieves all products from the in-memory list and responds with the list in JSON format.
5. ****POST Endpoint****:
 - The `/products` route accepts a JSON payload to create a new `Product` and adds it to the in-memory list. It returns the created product with a `201 Created` status code.

Testing the API:

- Start the server (`localhost:8080`).
- ****GET Request****: Use a tool like Postman or cURL:


```
...
GET http://localhost:8080/products
...
Response: `[]` (initially empty).
```
- ****POST Request****: Add a new product:


```
...
POST http://localhost:8080/products
Content-Type: application/json
{
  "id": 1,
  "name": "Laptop",
  "price": 999.99
}
...
Response: `201 Created` with the created product.
```
- ****GET Request**** (again): Now you should see the added product in the response.

Response time: 7.01 seconds

Llama-3.1 Response:

Here's an example of how you can create a REST API using Ktor for the given data class:

Product Data Class and Repository

```
``kotlin
// data class
data class Product(val id: Int, val name: String, val price: Double)
// In-memory repository for simplicity
class ProductRepository {
    private val products: MutableList<Product> = mutableListOf()
    private var nextId: Int = 1
    fun getAllProducts(): List<Product> = products
    fun addProduct(name: String, price: Double): Product {
```

```

        val newProduct = Product(nextId++, name, price)
        products.add(newProduct)
        return newProduct
    }
}
...
### Ktor REST API
```kotlin
import io.ktor.application.*
import io.ktor.response.*
import io.ktor.request.*
import io.ktor.routing.*
import io.ktor.http.*
import io.ktor.gson.*
fun main(args: Array<String>): Unit = io.ktor.server.netty.EngineMain.main(args)
@Suppress("unused")
fun Application.module() {
 install(ContentNegotiation) {
 gson {
 setPrettyPrinting()
 }
 }
 val productRepository = ProductRepository()
 // Initialize some sample data
 productRepository.addProduct("Product 1", 9.99)
 productRepository.addProduct("Product 2", 19.99)
 routing {
 get("/products") {
 val products = productRepository.getAllProducts()
 call.respond(products)
 }
 post("/products") {
 val productRequest = call.receive<ProductRequest>()
 val newProduct = productRepository.addProduct(productRequest.name, productRequest.price)
 call.respond(HttpStatusCode.Created, newProduct)
 }
 }
}
}
data class ProductRequest(val name: String, val price: Double)
...

```

This example sets up a simple REST API with two endpoints:

1. **GET /products**: Returns a list of all products in the repository.
2. **POST /products**: Creates a new product and returns the created product.

The API uses Ktor's built-in support for JSON serialization/deserialization using Gson. The `ProductRequest` data class is used to deserialize the request body of the `POST /products` endpoint.

#### ### Example Use Cases

You can use a tool like `curl` to test the API endpoints:

```

```bash
# Get all products

```

```
curl http://localhost:8080/products
# Create a new product
curl -X POST -H "Content-Type: application/json" -d '{"name": "Product 3", "price": 29.99}'
http://localhost:8080/products
'''
```

Response time: 17.23 seconds

Few-shot Prompt:

```
Convert the data class to a REST API using Ktor:
data class Product(val id: Int, val name: String, val price: Double)
// TODO: Create GET and POST endpoints using Ktor
Example 1:
import io.ktor.application.*
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
val products = mutableListOf<Product>()
fun Application.module() {
    routing {
        get("/products") {
            call.respond(products)
        }
        get("/products/{id}") {
            val id = call.parameters["id"]?.toIntOrNull()
            val product = id?.let { products.find { p -> p.id == it } }
            if (product != null) call.respond(product) else call.respondText("Not Found", status =
HttpStatusCode.NotFound)
        }
        post("/products") {
            val product = call.receive<Product>()
            products.add(product)
            call.respondText("Product added", status = HttpStatusCode.Created)
        }
    }
}
Example 2:
import io.ktor.application.*
import io.ktor.features.*
import io.ktor.http.*
import io.ktor.jackson.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
class ProductService {
    private val products = mutableMapOf<Int, Product>()

    fun getAll(): List<Product> = products.values.toList()
```

```

fun getByld(id: Int): Product? = products[id]
fun add(product: Product): Product {
    products[product.id] = product
    return product
}
}
fun Application.configureRouting() {
    val service = ProductService()
    install(ContentNegotiation) {
        jackson {}
    }
    routing {
        route("/api/products") {
            get {
                call.respond(service.getAll())
            }
            get("/{id}") {
                val id = call.parameters["id"]?.toIntOrNull() ?: return@get call.respond(HttpStatusCode.BadRequest)
                val product = service.getByld(id) ?: return@get call.respond(HttpStatusCode.NotFound)
                call.respond(product)
            }
            post {
                val product = call.receive<Product>()
                service.add(product)
                call.respond(HttpStatusCode.Created, product)
            }
        }
    }
}
}

```

GPT-4o Response:

Here's how you can convert your `Product` data class into a REST API using Ktor, incorporating both `GET` and `POST` endpoints. I'll combine aspects of the two examples and provide a clean and production-ready implementation.

Full Implementation

```

```kotlin
import io.ktor.application.*
import io.ktor.features.*
import io.ktor.http.*
import io.ktor.jackson.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.server.engine.*
import io.ktor.server.netty.*

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

class ProductService {

```



```

private val products = mutableMapOf<Int, Product>()
fun getAll(): List<Product> = products.values.toList()
fun getById(id: Int): Product? = products[id]
fun add(product: Product): Product {
 products[product.id] = product
 return product
}
}

fun Application.module() {
 val productService = ProductService()

 // Install JSON support using Jackson
 install(ContentNegotiation) {
 jackson {}
 }

 routing {
 route("/api/products") {
 // Get all products
 get {
 call.respond(productService.getAll())
 }
 // Get a single product by ID
 get("/{id}") {
 val id = call.parameters["id"]?.toIntOrNull()
 if (id == null) {
 call.respond(HttpStatusCode.BadRequest, "Invalid or missing ID")
 return@get
 }

 val product = productService.getById(id)
 if (product == null) {
 call.respond(HttpStatusCode.NotFound, "Product not found")
 } else {
 call.respond(product)
 }
 }
 }

 // Add a new product
 post {
 try {
 val product = call.receive<Product>()
 if (productService.getById(product.id) != null) {
 call.respond(HttpStatusCode.Conflict, "Product with ID ${product.id} already exists")
 } else {
 productService.add(product)
 call.respond(HttpStatusCode.Created, product)
 }
 } catch (e: Exception) {
 call.respond(HttpStatusCode.BadRequest, "Invalid product data")
 }
 }
 }
}

```

```

 }
 }
}
}

// Entry point to start the server
fun main() {
 embeddedServer(Netty, port = 8080, module = Application::module).start(wait = true)
}
...

Explanation:
1. Data Class:
 - The `Product` data class has fields for `id`, `name`, and `price`.
2. Service Layer:
 - Introduced a `ProductService` class to encapsulate logic for managing products. It uses an in-memory `MutableMap` to store products.
3. Routing:
 - `/api/products` (GET): Fetch all products.
 - `/api/products/{id}` (GET): Fetch a single product by its ID. Returns `400 Bad Request` if the ID is invalid or missing, or `404 Not Found` if the product doesn't exist.
 - `/api/products` (POST): Add a new product. Returns `409 Conflict` if a product with the same ID already exists.
4. Error Handling:
 - Proper error handling for scenarios like invalid input, missing parameters, or conflicts during product creation.
 - Returns appropriate HTTP status codes (`400`, `404`, `409`, etc.).
5. JSON Support:
 - Installed the `ContentNegotiation` feature with Jackson to handle JSON serialization and deserialization automatically.
6. Server Setup:
 - Used `embeddedServer` with Netty to start the server on port 8080.

Testing the API:
1. Start the Server
Run the `main` function, which starts the server on `http://localhost:8080`.
2. Test Endpoints
You can use tools like [Postman](https://www.postman.com/) or `curl` to test the API.
Example Requests:
- Get All Products:
    ```bash
    curl http://localhost:8080/api/products
    ```
- Get Product by ID:
    ```bash
    curl http://localhost:8080/api/products/1
    ```
- Add a Product:

```

```
```bash
  curl -X POST -H "Content-Type: application/json" -d '{"id":1,"name":"Laptop","price":1200.0}'
http://localhost:8080/api/products
```
```

---  
This implementation provides clear separation of concerns, handles edge cases, and follows best practices for building REST APIs with Ktor. You can further extend it with database integration, authentication, or additional endpoints as needed.

Response time: 19.07 seconds

Llama-3.1 Response:

Below is an example of how to create a REST API using Ktor for the provided data class.

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

```kotlin
// ProductController.kt
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*

class ProductController(private val service: ProductService) {
    suspend fun getAll(call: ApplicationCall) {
        call.respond(service.getAll())
    }
    suspend fun getById(call: ApplicationCall) {
        val id = call.parameters["id"]?.toIntOrNull() ?: return call.respond(HttpStatusCode.BadRequest)
        val product = service.getById(id) ?: return call.respond(HttpStatusCode.NotFound)
        call.respond(product)
    }
    suspend fun add(call: ApplicationCall) {
        val product = call.receive<Product>()
        val addedProduct = service.add(product)
        call.respond(HttpStatusCode.Created, addedProduct)
    }
}
```

```kotlin
// ProductService.kt
class ProductService {
    private val products = mutableMapOf<Int, Product>()
    fun getAll(): List<Product> = products.values.toList()
    fun getById(id: Int): Product? = products[id]
    fun add(product: Product): Product {
        products[product.id] = product
        return product
    }
}
```
```

```

}
...
```kotlin
// Application.kt
import io.ktor.application.*
import io.ktor.features.*
import io.ktor.http.*
import io.ktor.jackson.*
import io.ktor.routing.*
fun Application.module() {
    val service = ProductService()
    val controller = ProductController(service)
    install(ContentNegotiation) {
        jackson {}
    }
    routing {
        route("/api/products") {
            get {
                controller.getAll(call)
            }
            get("/{id}") {
                controller.getById(call)
            }
            post {
                controller.add(call)
            }
        }
    }
}
...

```

In this code:

1. The `Product` class is defined.
2. The `ProductController` class is responsible for handling HTTP requests. It receives the `ProductService` instance in its constructor and uses it to interact with products.
3. The `ProductService` class encapsulates the product data and provides methods for retrieving and adding products.
4. The `Application.module` function configures the Ktor application, installing the Jackson content negotiation feature and defining routes for the product API.

Example use cases:

- To get all products: Send a GET request to `/api/products`
- To get a product by ID: Send a GET request to `/api/products/{id}`
- To add a new product: Send a POST request to `/api/products` with a JSON body containing the product details.

Note that this code assumes that you have the necessary dependencies in your `build.gradle` file:

```

```groovy
dependencies {
 implementation "io.ktor:ktor-server-core:$ktor_version"
 implementation "io.ktor:ktor-server-netty:$ktor_version"
 implementation "io.ktor:ktor-jackson:$ktor_version"
}

```

```
}
...
```

Response time: 20.36 seconds

**Analysis:** For Zero-Shot GPT-4o delivered a simple, complete Ktor API and used json for serialization, while Llama provided routing and used gson to serialize. With the examples of Few-Shot the jackson JSON processor is introduced. Both benefit from the examples, but I found GPT-4o's explanation was very helpful

**Comparison:** Use GPT-4o for standard solutions and detailed explanations and tradeoffs.

## Task 10: Function Summarization (Python)

Prompting Strategy 1: Zero-shot Prompt:

```
Write a brief summary of the function:
def reverse_words(sentence):
 return ' '.join(sentence.split()[::-1])
```

GPT-4o Response:

```
The `reverse_words` function takes a string input called `sentence`, splits it into a list of words, reverses the order of the words, and then joins them back together into a single string with spaces in between. It effectively returns the input sentence with the word order reversed.
```

Response time: 1.35 seconds

Llama-3.1 Response:

```
Function Summary: `reverse_words`
Reverses the order of words in a given sentence while maintaining the original word order within each word.
Parameters:
* `sentence` (str): The input sentence to be reversed.
Return Value:
* str: The input sentence with its words in reverse order.
Example:
```python  
print(reverse_words("Hello World")) # Outputs: "World Hello"  
```
```

Response time: 2.94 seconds

Few-shot Prompt:

```
Write a brief summary of the function:
def reverse_words(sentence):
 return ' '.join(sentence.split()[::-1])
Example 1:
This function reverses the order of words in a sentence while maintaining the original word order.
```

Example 2:

This function takes a sentence, splits it into words, reverses their order, and joins them back together with spaces, effectively reversing the sequence of words in the input sentence.

GPT-4o Response:

This function, ``reverse_words(sentence)``, takes a sentence as input, splits it into individual words, reverses the order of the words, and then joins them back together with spaces. The result is a new sentence with the words in reverse order while preserving the original order of characters within each word.

Response time: 1.33 seconds

Llama-3.1 Response:

```
Function Summary
`reverse_words(sentence)`
Reverses the order of words in a given sentence while preserving the original characters within each word.
Functionality:
- Splits the input sentence into individual words.
- Reverses the sequence of these words.
- Joins the reversed words back together with spaces, forming the output sentence.

Example Use Cases:
- `reverse_words("Hello World")` returns `"World Hello"`.
- `reverse_words("This is a test sentence")` returns `"sentence test a is This"`.
```

Response time: 3.87 seconds

**Analysis:** GPT-4o gave a concise summary ("This function reverses the order of words in a sentence"), while Llama included parameter and return value details. Both were accurate, with Llama offering more depth but it could be argued it strayed from being a "summary". With Few-Shot examples, both responses get wordier, but Llama goes too far.

**Comparison:** Choose GPT-4o for brevity. Llama was too verbose.

## Task 11: Prompt from Code Comments (Python)

Prompting Strategy 1: Zero-shot Prompt:

```
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
```

GPT-4o Response:

**\*\*Prompt:\*\***

"Write a Python function named `is\_prime` that determines if a given integer is a prime number. The function should take one argument, `n`, which is the number to check. A number is considered prime if it is greater than 1 and divisible only by 1 and itself. Use an efficient algorithm to minimize the number of iterations, such as checking divisors up to the square root of the number. Return `True` if the number is prime, and `False` otherwise."

Response time: 1.66 seconds

Llama-3.1 Response:

Here's a prompt that could generate the code:

"Write a function in Python that checks if a given integer is a prime number. The function should return True if the number is prime and False otherwise. Ensure the function is efficient by only checking divisibility up to the square root of the number."

Response time: 2.23 seconds

Few-shot Prompt:

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
```

Example 1:

Write a Python function named `is_prime` that determines if a given number is prime. The function should return False for numbers less than or equal to 1, and check divisibility from 2 up to the square root of the input number for efficiency.

Example 2:

Implement a function called `is_prime` in Python that checks whether a number is prime. A prime number is only divisible by 1 and itself. For efficiency, the function should only check divisors up to the square root of the input number, and should handle edge cases like numbers less than 2.

GPT-4o Response:

"Write a Python function named `is\_prime` that determines whether a given number is prime. The function should return `False` for numbers less than or equal to 1 and only check potential divisors up to the square root of the given number for efficiency. A prime number is defined as a number greater than 1 that is only divisible by 1 and itself. Include edge case handling and ensure the implementation is efficient."

Response time: 1.75 seconds

Llama-3.1 Response:

Here are the prompts that can generate the code:

1. "Write a Python function named `is_prime` that checks whether a given integer is prime, returns False for numbers less than or equal to 1, and checks divisibility from 2 up to the square root of the input number for efficiency."
2. "Generate a Python function called `is_prime` that determines if a number is prime, handling edge cases for numbers less than 2 and optimizing by only checking divisors up to the square root of the input number."
3. "Implement an efficient Python function `is_prime` that checks whether a number is prime by testing divisibility up to the square root of the number, and returns False for input numbers less than or equal to 1."
4. "Create a Python function named `is_prime` that uses a loop to check divisibility from 2 up to the square root of the input number to determine if the number is prime, and returns False for numbers less than or equal to 1."
5. "Write an optimized Python function `is_prime` that checks if a number is prime by only testing divisors up to the square root of the number, and handles edge cases for numbers less than 2."

These prompts clearly specify the requirements of the function, including the name, input, output, edge cases, and optimization, which can help generate the code accurately.

Response time: 9.73 seconds

**Analysis:** With Zero-Prompt GPT-4o crafted a detailed prompt which included number of parameters, definition of prime, directions to ensure efficiency, while Llama surprisingly generated a concise prompt. Both met the requirements effectively. Few-Shot explicitly calls on the models to handle edge cases. GPT-4o tacks that onto its existing prompt, but Llama goes nuts and generates 5 different responses. Well, this gives you options.

**Comparison:** Use GPT-4o for precise prompts or Llama for flexible options.

## Task 12: Fixing Factorial Bug (Python)

Prompting Strategy 1: Zero-shot Prompt:

Fix the bug when input is 0:

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

GPT-4o Response:

There is a bug in the implementation of the `factorial` function. Specifically, the function does not correctly handle the case where `n = 0`. In mathematics, the factorial of 0 (`0!`) is defined as 1. Additionally, the loop in the function is set to iterate from 1 to `n-1`, which skips the last term (`n`) altogether.

Here is the corrected version of the `factorial` function:



```

```python
def factorial(n):
    if n == 0:
        return 1 # 0! is defined as 1
    result = 1
    for i in range(1, n + 1): # Fix the loop to include n
        result *= i
    return result
```

Explanation of Fixes:
1. **Handle the case when `n == 0`:**
 - Added a condition at the start of the function to return `1` if the input `n` is `0`.

2. **Correct the range in the loop:**
 - Changed the loop range from `range(1, n)` to `range(1, n + 1)`, so the loop includes `n` in the multiplication. The original implementation incorrectly stopped at `n - 1`.

Example Usage:
```python
print(factorial(0)) # Output: 1
print(factorial(1)) # Output: 1
print(factorial(5)) # Output: 120
```

```

Response time: 48.41 seconds

Llama-3.1 Response:

The issue with the given code is that it does not handle the case when the input is 0 correctly. In mathematics, the factorial of 0 is defined as 1. However, the given code will return 1 for an input of 0, but it will do so for the wrong reason. The loop will not execute at all, and it simply returns the initial value of `result`, which is 1.

However, there's another bug in the code. The loop should iterate up to `n` (inclusive), not up to `n-1`. This is because the `range` function in Python generates numbers up to but not including the stop value.

Here's the corrected code:

```

```python
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    Args:
        n (int): The input number.

    Returns:
    """

```

int: The factorial of n.

Raises:

ValueError: If n is a negative integer.

"""

if not isinstance(n, int):

raise TypeError("Input must be a non-negative integer.")

if n < 0:

raise ValueError("Input must be a non-negative integer.")

elif n == 0 or n == 1:

return 1

else:

result = 1

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

result *= i

return result

"""

Example use cases:

```python

print(factorial(0)) # Output: 1

print(factorial(1)) # Output: 1

print(factorial(5)) # Output: 120

print(factorial(-1)) # Raises ValueError

print(factorial(1.5)) # Raises TypeError

```

Response time: 12.46 seconds

Few-shot Prompt:

Fix the bug when input is 0:

def factorial(n):

result = 1

for i in range(1, n):

result *= i

return result

Example 1:

The bug is that when n=0, the function returns 1 without multiplying anything, which is correct for factorial(0)=1, but the loop range is wrong. The function should use range(1, n+1) to include n itself:

def factorial(n):

result = 1

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

result *= i

return result

Example 2:

There are two issues: 1) The range should be (1, n+1) to include n itself, and 2) The function correctly returns 1 for n=0 by default, which matches the mathematical definition of $0! = 1$. Here's the fixed version:

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

GPT-4o Response:

Your observations about the issues in the original code and the fixes are correct. Let me clarify and finalize the explanation for fixing the bug when `n=0`:

Issue in the Original Code:

1. When `n=0`, the loop `for i in range(1, n)` does not execute because `range(1, 0)` is empty. However, the function still works correctly for `factorial(0)` because the `result` is initialized to `1`, which is mathematically correct ($0! = 1$).

2. The loop range is also incorrect for values of `n > 0`, as it does not include the upper bound `n` (it only iterates up to `n-1`).

Correct Fix:

To fix the loop so it includes `n` itself and works correctly for all non-negative integers, we need to change the range to `(1, n+1)`. This ensures that the loop iterates over all integers from `1` to `n` inclusive.

Here is the corrected code:

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

Explanation of Changes:

1. **Range Fix**: The loop now uses `range(1, n+1)`, which ensures all integers from `1` to `n` are included in the multiplication.

2. **Default Return for `n=0`**: The function correctly returns `1` for `n=0` because the loop does not execute, and the initial value of `result` is `1`.

Examples:

Let's test the corrected function with some examples:

```
```python
print(factorial(0)) # Output: 1 (0! = 1)
```

```
print(factorial(1)) # Output: 1 (1! = 1)
print(factorial(5)) # Output: 120 (5! = 5*4*3*2*1 = 120)
...
```

### Edge Cases:

- `n=0`: Returns `1` (correctly handles the mathematical definition of  $0! = 1$ ).
- `n=1`: Returns `1` ( $1! = 1$ ).
- Larger values of `n`: The range and multiplication logic correctly compute the factorial.

This fixed version handles all edge cases and works correctly for all non-negative integers.

Response time: 8.12 seconds

Llama-3.1 Response:

The code is almost correct, but it still doesn't handle the case when the input is a negative number. Factorial is only defined for non-negative integers, so we should add a check to handle this.

Here's the fixed version:

```
```python
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    Args:
    n (int): The input number.

    Returns:
    int: The factorial of n.

    Raises:
    ValueError: If n is a negative integer.
    TypeError: If n is not an integer.
    """
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```
```

This code correctly calculates the factorial of a non-negative integer, and raises an error for negative integers and non-integer inputs. The `+1` in `range(1, n + 1)` is necessary to include `n` itself in the loop.

Example usage:

```
```python
print(factorial(5)) # Output: 120
print(factorial(0)) # Output: 1
print(factorial(-1)) # Raises ValueError
print(factorial(1.5)) # Raises TypeError
```
```

Response time: 610.32 seconds

**Analysis:** GPT-4o fixed the range (`range(1, n + 1)`) and handled `n=0`, while Llama went further and checked to see if the number was non-negative, an integer (not fraction or 0), or 1. They both corrected for the loop to range from 1 to `n+1`. Prompting helped GPT minimally and only served to tighten up the already robust Llama code.

**Comparison:** Select Llama for robust code, but it did take quite a long time 610.32 seconds.

### Task 13: Linked List Node Deletion (C)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

Here is the implementation for the `deleteNode` function to delete a node by value (`key`) from a linked list:

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

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

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

    // If head node itself holds the key to be deleted
```

```

if (temp != NULL && temp->data == key) {
    *head = temp->next; // Change head to the next node
    free(temp);        // Free memory of the old head
    return;
}

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

// If the key was not present in the linked list
if (temp == NULL) {
    printf("Key %d not found in the list.\n", key);
    return;
}

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

// Free memory of the node to be deleted
free(temp);
}

// Helper function to add a new node to the end of the linked list
void appendNode(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* last = *head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
}

// Helper function to print the linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
}

```

```

}
printf("NULL\n");
}

// Example usage
int main() {
    struct Node* head = NULL;

    appendNode(&head, 10);
    appendNode(&head, 20);
    appendNode(&head, 30);
    appendNode(&head, 40);
    appendNode(&head, 50);

    printf("Original List: ");
    printList(head);

    deleteNode(&head, 20);
    printf("After deleting 20: ");
    printList(head);

    deleteNode(&head, 10);
    printf("After deleting 10: ");
    printList(head);

    deleteNode(&head, 50);
    printf("After deleting 50: ");
    printList(head);

    deleteNode(&head, 100); // Key not in the list
    printf("After trying to delete 100: ");
    printList(head);

    return 0;
}
...

```

Explanation:

1. ****Special Case for Head Node****:

- If the head node itself contains the key, update the head to point to the next node, and free the memory of the original head.

2. ****Traverse the List****:

- Use a `prev` pointer to keep track of the previous node while searching for the node containing the key.

3. ****Key Not Found****:

- If the traversal ends and the key is not found (`temp == NULL`), print a message and exit the function.

4. ****Unlink the Node****:
 - If the key is found, update the `next` pointer of the `prev` node to skip over the node to be deleted.
5. ****Free the Memory****:
 - Free the memory allocated for the node to be deleted to avoid memory leaks.

Output Example:

```
...
Original List: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
After deleting 20: 10 -> 30 -> 40 -> 50 -> NULL
After deleting 10: 30 -> 40 -> 50 -> NULL
After deleting 50: 30 -> 40 -> NULL
Key 100 not found in the list.
After trying to delete 100: 30 -> 40 -> NULL
...
```

Response time: 24.68 seconds

Llama-3.1 Response:

Here's an implementation of the `deleteNode` function that deletes a node with a given value from a singly linked list:

```
```c
/**
 * Deletes a node with the given key from the linked list.
 *
 * @param head A pointer to the head of the linked list.
 * @param key The value of the node to be deleted.
 */
void deleteNode(struct Node** head, int key) {
 // Check if the linked list is empty
 if (*head == NULL) return;

 // If the node to be deleted is the head node
 if ((*head)->data == key) {
 struct Node* temp = *head;
 *head = (*head)->next;
 free(temp);
 return;
 }

 // Traverse the linked list to find the node to be deleted
 struct Node* current = *head;
 while (current->next != NULL) {
 if (current->next->data == key) {
 struct Node* temp = current->next;
 current->next = current->next->next;
 free(temp);
 return;
 }
 current = current->next;
 }
}
```



```

 }
 current = current->next;
}
}
...

```

Here's an explanation of the implementation:

1. We first check if the linked list is empty by checking if `\*head` is `NULL`. If it is, we return immediately.
2. We then check if the node to be deleted is the head node by comparing `(\*head)->data` with the given `key`. If it is, we update `\*head` to point to the next node in the list and free the memory allocated for the head node.
3. If the node to be deleted is not the head node, we traverse the linked list to find the node with the given `key`. We do this by iterating through the list using a `while` loop and checking if the data of the next node matches the `key`.
4. When we find the node to be deleted, we update the `next` pointer of the previous node to skip over the node to be deleted and free the memory allocated for the node.
5. If we reach the end of the linked list without finding the node to be deleted, we simply return without doing anything.

Example use case:

```

```c
int main() {
    struct Node* head = NULL;

    // Insert some nodes into the linked list
    head = malloc(sizeof(struct Node));
    head->data = 1;
    head->next = malloc(sizeof(struct Node));
    head->next->data = 2;
    head->next->next = malloc(sizeof(struct Node));
    head->next->next->data = 3;
    head->next->next->next = NULL;

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

    // Print the remaining nodes in the linked list
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");

    return 0;
}
...

```

This code will output `1 3`, which are the remaining nodes in the linked list after deleting the node with value 2.

Response time: 24.41 seconds

Few-shot Prompt:

Implement node deletion by value:

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

Example 1:

```
void deleteNode(struct Node** head, int key) {  
    struct Node *temp = *head, *prev = NULL;  
  
    // If head node itself holds the key  
    if (temp != NULL && temp->data == key) {  
        *head = temp->next;  
        free(temp);  
        return;  
    }  
  
    // Search for the key, keep track of previous node  
    while (temp != NULL && temp->data != key) {  
        prev = temp;  
        temp = temp->next;  
    }  
  
    // If key was not present  
    if (temp == NULL) return;  
  
    // Unlink the node from linked list  
    prev->next = temp->next;  
    free(temp);  
}
```

Example 2:

```
void deleteNode(struct Node** head, int key) {  
    // Handle empty list  
    if (*head == NULL) return;  
  
    struct Node* current = *head;  
  
    // Special case: delete head node  
    if (current->data == key) {  
        *head = current->next;  
        free(current);  
    }
```

```

    return;
}

// Search for the node
while (current->next != NULL && current->next->data != key) {
    current = current->next;
}

// If found, delete it
if (current->next != NULL) {
    struct Node* temp = current->next;
    current->next = temp->next;
    free(temp);
}
}

```

GPT-4o Response:

Here is the implementation of the `deleteNode` function to delete a node by its value from a linked list. The function takes a pointer to the head of the linked list and the value (`key`) to be deleted. It handles all possible scenarios, such as the value being in the head node, somewhere in the middle, or not being present in the list at all.

Full Implementation:

```

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

// Definition of the linked list node
struct Node {
 int data;
 struct Node* next;
};

// Function to delete a node by value
void deleteNode(struct Node** head, int key) {
 // If the list is empty
 if (*head == NULL) {
 return;
 }

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

 // Case 1: The head node itself holds the key
 if (temp != NULL && temp->data == key) {
 *head = temp->next; // Move the head pointer to the next node
 free(temp); // Free the old head node
 }
}

```

```

 return;
}

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

// Case 3: If the key was not found in the list
if (temp == NULL) {
 return; // Key is not present in the linked list
}

// Case 4: Unlink the node from the linked list
prev->next = temp->next;

// Free the memory of the node to be deleted
free(temp);
}

// Utility function to create a new node
struct Node* newNode(int data) {
 struct Node* node = (struct Node*)malloc(sizeof(struct Node));
 node->data = data;
 node->next = NULL;
 return node;
}

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

// Test the deleteNode function
int main() {
 // Create a linked list: 10 -> 20 -> 30 -> 40 -> 50
 struct Node* head = newNode(10);
 head->next = newNode(20);
 head->next->next = newNode(30);
 head->next->next->next = newNode(40);
 head->next->next->next->next = newNode(50);

 printf("Original list:\n");

```

```

printList(head);

// Delete node with value 30
printf("\nDeleting node with value 30...\n");
deleteNode(&head, 30);
printList(head);

// Delete node with value 10 (head node)
printf("\nDeleting node with value 10...\n");
deleteNode(&head, 10);
printList(head);

// Delete node with value 50 (last node)
printf("\nDeleting node with value 50...\n");
deleteNode(&head, 50);
printList(head);

// Attempt to delete a value not in the list
printf("\nDeleting node with value 100 (not in list)...\n");
deleteNode(&head, 100);
printList(head);

return 0;
}
....

```

#### ### Explanation of the Implementation:

1. **Empty List:** If the list is empty (`*head == NULL`), the function simply returns without doing anything.
2. **Head Node Deletion:** If the head node contains the key, update the head pointer to point to the next node, then free the memory of the old head node.
3. **Search and Delete:** Traverse the list to find the node with the given key. Keep track of the previous node so that the node can be unlinked when found.
4. **Key Not Found:** If the key does not exist in the list, the function exits without making any changes.
5. **Unlink and Free Node:** Once the node is found, unlink it from the list by updating the `next` pointer of the previous node, and free the memory occupied by the node.

#### ### Output Example:

For the above main function, the output will be:

....

Original list:

10 -> 20 -> 30 -> 40 -> 50 -> NULL

Deleting node with value 30...

10 -> 20 -> 40 -> 50 -> NULL

Deleting node with value 10...

20 -> 40 -> 50 -> NULL

Deleting node with value 50...

20 -> 40 -> NULL

Deleting node with value

Response time: 27.26 seconds

Llama-3.1 Response:

## ## Implementing Node Deletion by Value in a Linked List

Here's a high-quality implementation of the node deletion function in C, along with explanations and example use cases.

### ### Code

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

// Define the structure for a linked list node
struct Node {
    int data;
    struct Node* next;
};

/**
 * Deletes a node with the specified key from the linked list.
 *
 * @param head A pointer to the head of the linked list.
 * @param key The value of the node to be deleted.
 */
void deleteNode(struct Node** head, int key) {
    // Handle empty list
    if (*head == NULL) return;

    // Special case: delete head node
    if ((*head)->data == key) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);
        return;
    }

    // Search for the node
    struct Node* current = *head;
    while (current->next != NULL && current->next->data != key) {
        current = current->next;
    }
}
```

```

// If found, delete it
if (current->next != NULL) {
    struct Node* temp = current->next;
    current->next = temp->next;
    free(temp);
}
}

// Helper function to insert a new node at the end of the list
void insertNode(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* current = *head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = newNode;
}

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

int main() {
    // Create a sample linked list: 1 -> 2 -> 3 -> 4 -> 5
    struct Node* head = NULL;
    insertNode(&head, 1);
    insertNode(&head, 2);
    insertNode(&head, 3);
    insertNode(&head, 4);
    insertNode(&head, 5);

    printf("Original List: ");
    printList(head);
}

```

```

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

printf("List after deleting node with key 3: ");
printList(head);

// Delete node with key 1 (head node)
deleteNode(&head, 1);

printf("List after deleting node with key 1: ");
printList(head);

// Delete node with key 6 (non-existent node)
deleteNode(&head, 6);

printf("List after deleting node with key 6: ");
printList(head);

return 0;
}
...

```

Explanation

The `deleteNode` function takes a pointer to the head of the linked list and the key of the node to be deleted as input. It first checks if the head node itself holds the key, in which case it updates the head pointer and frees the memory allocated to the head node.

If the head node does not hold the key, the function traverses the linked list until it finds the node with the specified key or reaches the end of the list. If the node is found, it updates the `next` pointer of the previous node to skip the node to be deleted and frees the memory allocated to the deleted node.

The `insertNode` function is a helper function that inserts a new node at the end of the linked list, and the `printList` function is a helper function that prints the elements of the linked list.

In the `main` function, we create a sample linked list and demonstrate the usage of the `deleteNode` function by deleting nodes with keys 3, 1, and 6. The output of the program shows the original list and the list after each deletion operation.

Response time: 30.02 seconds

Analysis: Both models handled the linked list deletion algorithm correctly. GPT-4o provided more than asked for with an appendNode and printList functions. These extra functions are used in the setup and verification of the deleteNode function. GPT also prints if the node was not found. Llama included more extensive comments and explicit memory management considerations. After the examples in Few-Shot, GPT removes the appendNode and explicitly does this in the main. It does add a newNode function (good idea to encapsulate). Llama introduces insertNode (which allocates the memory like addNode) and printList.

Comparison: I like the Few-Shot Llama version best because it encapsulates the malloc and avoids the bizarre node appending GPT goes through in main after Few-Shot and its use of malloc in the main after Zero-Prompt.

Task 14: Recursive Function Completion (Python)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

Sure! Below is the completed recursive function for calculating the Fibonacci sequence:

```
```python
def fibonacci(n):
 # Base cases
 if n == 0:
 return 0
 elif n == 1:
 return 1

 # Recursive call
 return fibonacci(n - 1) + fibonacci(n - 2)
```
```

Explanation:

- **Base cases**:

- If `n == 0`, the Fibonacci number is `0`.
- If `n == 1`, the Fibonacci number is `1`.

- **Recursive call**:

- For `n > 1`, the Fibonacci number is the sum of the two preceding numbers, which is calculated as `fibonacci(n - 1) + fibonacci(n - 2)`.

Example:

```
```python
print(fibonacci(0)) # Output: 0
print(fibonacci(1)) # Output: 1
print(fibonacci(5)) # Output: 5
print(fibonacci(10)) # Output: 55
```
```

Note:

This implementation has exponential time complexity ($O(2^n)$), as it recalculates values repeatedly. For large `n`, consider using **memoization** or an iterative approach to improve efficiency.

Response time: 4.26 seconds

Llama-3.1 Response:

Here's the completed recursive function for Fibonacci:

```
```python
def fibonacci(n):
 """
 Calculate the nth Fibonacci number using recursion.

 Args:
 n (int): The position of the Fibonacci number to calculate.

 Returns:
 int: The nth Fibonacci number.
 """
 # Base cases
 if n <= 0:
 raise ValueError("n must be a positive integer")
 elif n == 1:
 return 0
 elif n == 2:
 return 1

 # Recursive call
 else:
 return fibonacci(n-1) + fibonacci(n-2)
```
```

****Example use case:****

```
```python
print(fibonacci(10)) # Output: 34
```
```

****Note:**** This recursive implementation has an exponential time complexity due to the repeated calculation of the same subproblems. For larger values of `n`, consider using an iterative approach or memoization to improve performance.

****Optimized version using memoization:****

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

 Args:
 n (int): The position of the Fibonacci number to calculate.
 """
```

memo (dict, optional): A dictionary to store previously calculated Fibonacci numbers. Defaults to {}.

Returns:

int: The nth Fibonacci number.

"""

# Base cases

if n <= 0:

raise ValueError("n must be a positive integer")

elif n == 1:

return 0

elif n == 2:

return 1

# Check if the Fibonacci number is already calculated

if n in memo:

return memo[n]

# Recursive call with memoization

else:

result = fibonacci(n-1, memo) + fibonacci(n-2, memo)

memo[n] = result

return result

...

This optimized version has a linear time complexity, making it suitable for larger values of `n`.

Response time: 24.85 seconds

Few-shot Prompt:

Complete the recursive function for Fibonacci:

def fibonacci(n):

# TODO: Base cases and recursive call

pass

Example 1:

def fibonacci(n):

if n <= 0:

return 0

elif n == 1:

return 1

else:

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

Example 2:

def fibonacci(n):

# Base cases

if n == 0:

return 0

if n == 1 or n == 2:

```
 return 1
Recursive call
return fibonacci(n-1) + fibonacci(n-2)
```

GPT-4o Response:

Here's the completed recursive function for the Fibonacci sequence based on your examples:

### Explanation

The Fibonacci sequence is defined as:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$  for  $n > 1$

The function uses base cases for  $n = 0$  and  $n = 1$ , and for all other  $n$ , it computes the Fibonacci number using a recursive call.

Here's the completed code:

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

### Example Usage

You can test the function as follows:

```
```python
# Test cases
print(fibonacci(0)) # Output: 0
print(fibonacci(1)) # Output: 1
print(fibonacci(2)) # Output: 1
print(fibonacci(3)) # Output: 2
print(fibonacci(4)) # Output: 3
print(fibonacci(5)) # Output: 5
print(fibonacci(6)) # Output: 8
print(fibonacci(7)) # Output: 13
```
```

### Notes:

1. While the recursive implementation is simple and elegant, it has exponential time complexity,  $O(2^n)$ , because it recalculates the same Fibonacci numbers multiple times.

2. For large values of  $n$ , consider using **memoization** (e.g., with a dictionary or `functools.lru_cache`) or an **iterative approach** to optimize performance.

Here's an optimized version using memoization:

```
```python
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive call with memoization
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]
```
```

Alternatively, you can use Python's `functools.lru_cache` decorator to achieve the same effect:

```
```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive call
    return fibonacci(n-1) + fibonacci(n-2)
```
```

Response time: 6.58 seconds

Llama-3.1 Response:

Here's a complete recursive function for Fibonacci with proper documentation and handling:

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

    Args:
        n (int): The position of the Fibonacci number to calculate.
    """
    if n < 0:
        raise ValueError("n must be a non-negative integer")
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```
```

Returns:

int: The nth Fibonacci number.

Raises:

TypeError: If n is not an integer.

ValueError: If n is a negative integer.

Example:

```
>>> fibonacci(10)
```

```
55
```

```
"""
```

```
Check if input is a non-negative integer
```

```
if not isinstance(n, int):
```

```
 raise TypeError("Input must be an integer.")
```

```
if n < 0:
```

```
 raise ValueError("Input must be a non-negative integer.")
```

```
Base cases
```

```
if n == 0:
```

```
 return 0
```

```
elif n == 1 or n == 2:
```

```
 return 1
```

```
Recursive call
```

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

```
'''
```

However, this recursive implementation has an exponential time complexity due to the repeated calculation of the same Fibonacci numbers. For larger inputs, it's recommended to use an iterative approach or a recursive approach with memoization to improve performance.

Here's an example of a recursive Fibonacci function with memoization:

```
```python
```

```
def fibonacci(n, memo={}):
```

```
    """
```

```
    Calculate the nth Fibonacci number.
```

Args:

n (int): The position of the Fibonacci number to calculate.

memo (dict): A dictionary to store the Fibonacci numbers that have been calculated.

Returns:

int: The nth Fibonacci number.

Raises:

TypeError: If n is not an integer.

ValueError: If n is a negative integer.

Example:

```
>>> fibonacci(10)
55
"""
# Check if input is a non-negative integer
if not isinstance(n, int):
    raise TypeError("Input must be an integer.")
if n < 0:
    raise ValueError("Input must be a non-negative integer.")

# Base cases
if n == 0:
    return 0
elif n == 1 or n == 2:
    return 1

# Check if Fibonacci number has been calculated
if n in memo:
    return memo[n]

# Recursive call
result = fibonacci(n-1, memo) + fibonacci(n-2, memo)

# Store the calculated Fibonacci number
memo[n] = result

return result
'''
```

This implementation has a linear time complexity, making it much more efficient for larger inputs.

Response time: 17.14 seconds

Analysis: For Zero-Shot only GPT-4o correctly implemented the function. Probably because it did not account for $\text{fib}(0) = 0$, but rather raised an error, Llama calculated $\text{Fib}(10)=34$ which is wrong, though it corrected itself in the few shot prompt. Furthermore, it went on to implement a memory optimized version which was unasked for. For Few-Prompt GPT-4o's failed to include the less than 0 or integers only input and then went on to add the memory optimized solution, including the use of `lru_cache`. Llama has generated great comments, raises errors for non-integers, now handles $\text{fib}(0)$ correctly and also implemented a memory efficient version. Additionally Llama explained the recursion logic in detail using inline comments.

Comparison: Choose GPT-4o for efficient code or Llama with prompts for learning recursion.

Task 15: Constructor Completion (Python)

Prompting Strategy 1: Zero-shot Prompt:

Complete the class constructor:

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

GPT-4o Response:

Here's how you can complete the constructor for the `Person` class to include `name`, `age`, and an optional `email`:

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

Explanation:

1. `**name` and `age`: These are required arguments, so they must be passed when creating a `Person` instance.
2. `**email`: This is optional because it has a default value of `None`. If no email is provided when creating an instance, it will default to `None`.
3. `**self`: Refers to the instance of the class, and `self.name`, `self.age`, and `self.email` are instance variables.

Example Usage:

```
```python
Creating a person with name and age only
person1 = Person("Alice", 30)
print(person1.name) # Output: Alice
print(person1.age) # Output: 30
print(person1.email) # Output: None

Creating a person with name, age, and email
person2 = Person("Bob", 25, "bob@example.com")
print(person2.name) # Output: Bob
print(person2.age) # Output: 25
print(person2.email) # Output: bob@example.com
```
```

Response time: 4.93 seconds

Llama-3.1 Response:

Here's a Python class constructor that includes name, age, and optional email.

```
```python
class Person:
```



```
def __init__(self, name: str, age: int, email: str = None):
 """
 Initializes a Person object.

 Args:
 name (str): The person's name.
 age (int): The person's age.
 email (str, optional): The person's email. Defaults to None.
 """
 self.name = name
 self.age = age
 self.email = email
...
```

In this constructor:

- `name` and `age` are required parameters.
- `email` is an optional parameter with a default value of `None`.

Here's how you can create a `Person` object using this class:

```
```python
# Create a person with email
person1 = Person("John Doe", 30, "john.doe@example.com")

# Create a person without email
person2 = Person("Jane Doe", 25)

# Print the attributes
print(person1.name) # Output: John Doe
print(person1.age)  # Output: 30
print(person1.email) # Output: john.doe@example.com

print(person2.name) # Output: Jane Doe
print(person2.age)  # Output: 25
print(person2.email) # Output: None
```
```

Response time: 9.23 seconds

Few-shot Prompt:

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

Example 1:

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

```
self.name = name
self.age = age
self.email = email
```

Example 2:

```
def __init__(self, name, age, email=""):
 if not isinstance(name, str):
 raise TypeError("Name must be a string")
 if not isinstance(age, int) or age < 0:
 raise ValueError("Age must be a non-negative integer")
 self.name = name
 self.age = age
 self.email = email
```

GPT-4o Response:

Here's a complete implementation of the `Person` class constructor that incorporates examples of both optional and validated attributes:

```
```python
class Person:
    def __init__(self, name, age, email=None):
        if not isinstance(name, str):
            raise TypeError("Name must be a string")
        if not isinstance(age, int) or age < 0:
            raise ValueError("Age must be a non-negative integer")
        if email is not None and not isinstance(email, str):
            raise TypeError("Email must be a string or None")

        self.name = name
        self.age = age
        self.email = email
```
```

### Explanation:

1. `name`:
  - Mandatory.
  - Must be a string, validated with `isinstance`.
  - If validation fails, a `TypeError` is raised.
2. `age`:
  - Mandatory.
  - Must be a non-negative integer.
  - If it's not an integer or is less than 0, a `ValueError` is raised.
3. `email`:
  - Optional. Defaults to `None` if not provided.
  - If provided and not `None`, it must be a string. Otherwise, a `TypeError` is raised.

This design ensures that the `Person` class has well-defined attributes while allowing flexibility for optional parameters.

Response time: 3.51 seconds

Llama-3.1 Response:

Here's the completed class constructor with input validation and documentation:

```
```python
class Person:
    """
    Represents a person with a name, age, and optional email.

    Attributes:
        name (str): The person's name.
        age (int): The person's age.
        email (str, optional): The person's email. Defaults to an empty string.
    """

    def __init__(self, name, age, email=""):
        """
        Initializes a Person object.

        Args:
            name (str): The person's name.
            age (int): The person's age.
            email (str, optional): The person's email. Defaults to an empty string.

        Raises:
            TypeError: If name is not a string.
            ValueError: If age is not a non-negative integer.
        """
        if not isinstance(name, str):
            raise TypeError("Name must be a string")
        if not isinstance(age, int) or age < 0:
            raise ValueError("Age must be a non-negative integer")
        if not isinstance(email, str):
            raise TypeError("Email must be a string")

        self.name = name
        self.age = age
        self.email = email
    ...

Example use cases:

```python
```

```

Create a person with name, age, and email
person1 = Person("John Doe", 30, "johndoe@example.com")

Create a person with name and age only
person2 = Person("Jane Doe", 25)

Attempt to create a person with invalid input
try:
 person3 = Person(123, 30) # Raises TypeError
except TypeError as e:
 print(e) # Output: Name must be a string

try:
 person4 = Person("John Doe", -1) # Raises ValueError
except ValueError as e:
 print(e) # Output: Age must be a non-negative integer
'''

```

Response time: 13.16 seconds

**Analysis:** Both initialized name, age, and email correctly. GPT-4o kept it simple, while Llama added type checking. With Few-Prompt GPT-4o begins to error check its inputs and Llama loses its parameter type checking. I think Llama should have kept the type checking. Overall they performed very similarly to each other.

**Comparison:** Use GPT-4o for basic implementations or Llama for validated code.

## Task 16: Binary Search Completion (Java)

Prompting Strategy 1: Zero-shot Prompt:

```

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

```

GPT-4o Response:

Here's the completed implementation of the binary search method:

```

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

```

```

int mid = left + (right - left) / 2; // Avoid potential overflow
if (arr[mid] == target) {
 return mid; // Target found, 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
}
...

```

#### ### Explanation:

##### 1. **Finding the middle index**:

- `int mid = left + (right - left) / 2`: This ensures no overflow occurs when calculating the midpoint, which could happen with `(left + right) / 2` for very large arrays.

##### 2. **Comparison and bounds adjustment**:

- If the middle element `arr[mid]` matches the `target`, return `mid`.
- If `arr[mid] < target`, the target must be in the right half, so adjust `left = mid + 1`.
- If `arr[mid] > target`, the target must be in the left half, so adjust `right = mid - 1`.

##### 3. **Return**:

- If the while loop exits without finding the target, return `-1` to indicate the target is not in the array.

#### ### Usage Example:

```

...java
public static void main(String[] args) {
 int[] arr = {1, 3, 5, 7, 9, 11};
 int target = 5;

 BinarySearchExample example = new BinarySearchExample();
 int result = example.binarySearch(arr, target);

 if (result != -1) {
 System.out.println("Target found at index " + result);
 } else {
 System.out.println("Target not found");
 }
}
...

```

This will output:

```

...
Target found at index 2
...

```

Response time: 5.76 seconds

Llama-3.1 Response:

Here's a complete implementation of binary search in Java:

```
```java
/**
 * Searches for a target value in a sorted array using binary search.
 *
 * @param arr    The sorted array to search in.
 * @param target The target value to search for.
 * @return The index of the target value if found, -1 otherwise.
 */
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2; // Avoids integer overflow

        // Compare the middle element with the target
        if (arr[mid] == target) {
            return mid; // Target found, return its index
        } else if (arr[mid] < target) {
            left = mid + 1; // Target is in the right half, adjust bounds
        } else {
            right = mid - 1; // Target is in the left half, adjust bounds
        }
    }
    return -1; // Target not found, return -1
}
```

Example Use Cases:

```java
public static void main(String[] args) {
    int[] sortedArray = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    BinarySearch bs = new BinarySearch();

    // Search for an element that exists in the array
    int target1 = 5;
    int result1 = bs.binarySearch(sortedArray, target1);
    System.out.println("Target " + target1 + " found at index " + result1);

    // Search for an element that doesn't exist in the array
    int target2 = 10;
    int result2 = bs.binarySearch(sortedArray, target2);
    System.out.println("Target " + target2 + " not found, result: " + result2);
}
```
```

\*\*\*  
\*\*Explanation:\*\*

1. We start with two pointers, `left` and `right`, representing the range of indices in the array that we're currently searching.
2. We calculate the middle index (`mid`) of this range using the formula  $\text{left} + (\text{right} - \text{left}) / 2$ . This avoids potential integer overflow issues when dealing with large arrays.
3. We compare the middle element (`arr[mid]`) with the target value (`target`).
  - \* If they match, we return the middle index, indicating that the target has been found.
  - \* If the middle element is less than the target, we know the target must be in the right half of the current range, so we adjust the `left` pointer to `mid + 1`.
  - \* If the middle element is greater than the target, we know the target must be in the left half of the current range, so we adjust the `right` pointer to `mid - 1`.
4. We repeat steps 2-3 until the `left` pointer exceeds the `right` pointer, indicating that the target is not in the array. In this case, we return -1 to indicate that the search was unsuccessful.

Response time: 23.35 seconds

Few-shot Prompt:

Complete the binary search implementation:

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

Example 1:

```
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;
}
```

Example 2:

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

```

while (left <= right) {
 int mid = left + (right - left) / 2; // Avoids integer overflow
 if (arr[mid] == target) {
 return mid; // Found target, return index
 }
 if (arr[mid] < target) {
 left = mid + 1; // Target is in right half
 } else {
 right = mid - 1; // Target is in left half
 }
}
return -1; // Target not found

```

GPT-4o Response:

Here's the completed implementation of the binary search algorithm. I'll also explain the differences and improvements between the examples provided.

### Final Implementation:

```

```java
public int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2; // Avoid integer overflow
        if (arr[mid] == target) {
            return mid; // Found target, return 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
}
```

```

### Explanation of Key Components:

1. **Initialization**:

- `left` and `right` are initialized to the bounds of the array: `left = 0` (start of the array), and `right = arr.length - 1` (end of the array).

2. **Midpoint Calculation**:

- `int mid = left + (right - left) / 2;` is used instead of `(left + right) / 2` to avoid potential integer overflow when `left` and `right` are large. This is a common best practice in binary search.

3. **Comparison Logic**:

- Compare the `arr[mid]` value with the target:
- If `arr[mid] == target`, the index `mid` is returned.



- If `arr[mid] < target`, the target must be in the right half, so `left = mid + 1`.
- If `arr[mid] > target`, the target must be in the left half, so `right = mid - 1`.

#### 4. **Termination**:

- The `while` loop continues as long as `left <= right`. If the loop exits, it means the target is not present in the array, and we return `-1`.

---

#### ### Example Usage:

##### #### Example 1:

```
```java
int[] arr = {2, 4, 6, 8, 10};
int target = 6;
System.out.println(binarySearch(arr, target)); // Output: 2 (index of 6)
```
```

##### #### Example 2:

```
```java
int[] arr = {1, 3, 5, 7, 9};
int target = 8;
System.out.println(binarySearch(arr, target)); // Output: -1 (8 is not in the array)
```
```

#### ### Complexity Analysis:

- **Time Complexity**:  $O(\log N)$ , where  $N$  is the size of the array. The search space is halved at each iteration.
- **Space Complexity**:  $O(1)$ , since the binary search operates in-place without requiring extra space.

This implementation is efficient, handles edge cases (e.g., empty arrays, large indices), and avoids integer overflow during midpoint calculations.

Response time: 8.02 seconds

#### Llama-3.1 Response:

Here is a completed binary search implementation in Java, along with explanations and example use cases:

```
```java
/**
 * Performs a binary search on a sorted array to find the index of a target element.
 *
 * @param arr the sorted array to search
 * @param target the element to find
 * @return the index of the target element if found, or -1 if not found
 */
public int binarySearch(int[] arr, int target) {
    int left = 0; // Left boundary of the search range
    int right = arr.length - 1; // Right boundary of the search range

    while (left <= right) {

```

```

// Calculate the midpoint of the search range, avoiding integer overflow
int mid = left + (right - left) / 2;

// Compare the midpoint element to the target
if (arr[mid] == target) {
    // Found the target, return its index
    return mid;
} else if (arr[mid] < target) {
    // Target is in the right half of the search range, adjust bounds
    left = mid + 1;
} else {
    // Target is in the left half of the search range, adjust bounds
    right = mid - 1;
}
}

// Target not found, return -1
return -1;
}
...

```

Example Use Cases

```

```java
public static void main(String[] args) {
 BinarySearch bs = new BinarySearch();

 // Example 1: Search for an element in a sorted array
 int[] arr1 = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
 int target1 = 23;
 int result1 = bs.binarySearch(arr1, target1);
 System.out.println("Target " + target1 + " found at index " + result1);

 // Example 2: Search for an element not in the array
 int[] arr2 = {1, 3, 5, 7, 9, 11, 13, 15};
 int target2 = 10;
 int result2 = bs.binarySearch(arr2, target2);
 System.out.println("Target " + target2 + " not found, result is " + result2);
}
...

```

#### ### Time Complexity

The time complexity of this binary search implementation is  $O(\log n)$ , where  $n$  is the length of the input array. This is because the algorithm divides the search range in half at each step, effectively reducing the number of comparisons needed to find the target element.

#### ### Space Complexity

The space complexity of this implementation is  $O(1)$ , as it only uses a constant amount of extra memory to store the left, right, and mid indices. The input array is not modified, and no additional data structures are allocated.

Response time: 22.18 seconds

**Analysis:** Both completed binary search correctly. GPT-4o's solution was minimal, while Llama included detailed comments on steps.

**Comparison:** Opt for GPT-4o for compact code or Llama for algorithmic understanding.

## Task 17: Self-Consistency Bug Fixing (C++)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

To resolve the inconsistency between the function name (`isOdd`) and its logic (which checks if a number is even), you can either:

1. **Rename the function to reflect its actual behavior:**

If the logic is correct and you intend to check for even numbers, rename the function to `isEven`:

```
```cpp  
// Correctly named function to reflect the logic  
bool isEven(int x) {  
    return x % 2 == 0;  
}  
...`
```

2. **Change the logic to match the original function name:**

If the function name `isOdd` is intentional, update the logic to correctly check for odd numbers:

```
```cpp  
// Correct logic to match the function name
bool isOdd(int x) {
 return x % 2 != 0;
}
...`
```

### Recommendation:

Choose the approach based on what the function is intended to do. If the goal is to check for even numbers, go with the first option. If the goal is to check for odd numbers, go with the second option.

Response time: 2.63 seconds

Llama-3.1 Response:

To resolve the inconsistency between the function name and its logic, you can either rename the function to match its logic or modify the logic to match the function name. Here are the two possible solutions:

**\*\*Solution 1: Rename the function\*\***

```
```cpp
// Renamed function to match its logic
bool isEven(int x) {
    return x % 2 == 0;
}
```
```

**\*\*Solution 2: Modify the logic\*\***

```
```cpp
// Modified logic to match the function name
bool isOdd(int x) {
    return x % 2 != 0;
}
```
```

In both cases, the function name and its logic are now consistent. Choose the solution that best fits your use case. If you intend to check for even numbers, use the first solution. If you intend to check for odd numbers, use the second solution.

Response time: 5.98 seconds

Few-shot Prompt:

Resolve inconsistency between function name and logic:

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

Example 1:

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

Example 2:

```
// Renamed function to match its behavior
bool isEven(int x) {
 return x % 2 == 0; // Returns true for even numbers
}

// Or alternatively, fix logic to match the name
bool isOdd(int x) {
 return x % 2 != 0; // Returns true for odd numbers
}
```

## GPT-4o Response:

The inconsistency in the original code arises because the function name `isOdd` implies it checks for odd numbers, but the logic inside the function actually determines whether a number is even. This contradiction can lead to confusion and bugs.

Here's how you can resolve it:

---

### ### \*\*Option 1: Fix the Function Name\*\*

Rename the function to match its existing logic (the function checks if the number is even, so the name should reflect that).

```cpp

// Adjusted function name to match its behavior

```
bool isEven(int x) {  
    return x % 2 == 0; // Returns true for even numbers  
}
```

```

---

### ### \*\*Option 2: Fix the Function Logic\*\*

Keep the function name `isOdd` but adjust the logic inside to correctly check for odd numbers.

```cpp

// Adjusted function logic to match its name

```
bool isOdd(int x) {  
    return x % 2 != 0; // Returns true for odd numbers  
}
```

```

---

### ### \*\*Which Option to Choose?\*\*

- If the intent of the function is to check for even numbers, **Option 1** is the correct choice.
- If the intent of the function is to check for odd numbers, **Option 2** is the correct choice.

Consistency between the function name and its behavior is critical for code readability and maintainability. Always ensure that the name of a function clearly reflects what it does.

Response time: 4.02 seconds

## Llama-3.1 Response:

To resolve the inconsistency between the function name and logic, you have two options:

### ### Option 1: Rename the function to match its behavior

```cpp

// Renamed function to match its behavior

```
bool isEven(int x) {  
    return x % 2 == 0; // Returns true for even numbers  
}
```

}

```
...  
### Option 2: Fix the logic to match the function name
```

```
```cpp  
// Fixed logic to match the function name
bool isOdd(int x) {
 return x % 2 != 0; // Returns true for odd numbers
}
...`
```

Both options will resolve the inconsistency, and the choice between them depends on the intended use of the function in your code.

```
Best Practice
```

To avoid such inconsistencies in the future, it's essential to follow best practices for naming functions and variables. The name should clearly reflect the purpose and behavior of the function or variable.

Additionally, including comments or documentation can help ensure that the code is understandable and maintainable, even for complex logic.

```
Example Use Cases
```

```
```cpp  
// Example usage of isEven function  
if (isEven(10)) {  
    std::cout << "10 is an even number." << std::endl;  
} else {  
    std::cout << "10 is an odd number." << std::endl;  
}  
  
// Example usage of isOdd function  
if (isOdd(11)) {  
    std::cout << "11 is an odd number." << std::endl;  
} else {  
    std::cout << "11 is an even number." << std::endl;  
}  
...`
```

Response time: 10.82 seconds

Analysis: Both fixed the naming/logic mismatch (e.g., renaming to isEven or adjusting logic). Both offered a quick fix, but Llama also suggested some use cases.

Comparison: Use GPT-4o for fast fixes or Llama for more verbose responses.

Task 18: Prompt Chaining: Bug Identification -> Fix (JavaScript)

Prompting Strategy 1: Zero-shot Prompt:

Identify and fix the bug:

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

GPT-4o Response:

The issue with the function is that it directly returns the result of `n % 2`. However, `n % 2` evaluates to `0` for even numbers and `1` for odd numbers, which are not equivalent to `true` and `false`. To fix this, you need to explicitly check if `n % 2` equals `0` and return the corresponding boolean value.

Here's the corrected version of the function:

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

Explanation of the Fix:

- `n % 2 === 0` checks if the remainder of `n` divided by `2` is `0`.
- If the remainder is `0`, the number is even, and the function will return `true`.
- If the remainder is not `0`, the number is odd, and the function will return `false`.

Example Usage:

```
```javascript  
console.log(isEven(4)); // true
console.log(isEven(7)); // false
console.log(isEven(0)); // true
console.log(isEven(-2)); // true
```
```

Response time: 3.90 seconds

Llama-3.1 Response:

Bug Identification

The bug in the given function `isEven(n)` is that it returns an integer value (0 or 1) instead of a boolean value (`true` or `false`). This can cause issues if the function is used in conditional statements or logical operations.

Debugged Code

To fix the bug, we can modify the function to return a boolean value by comparing the remainder of `n` divided by 2 to 0:

```
```javascript  
function isEven(n) {
```

```
return n % 2 === 0; // Returns true if n is even, false otherwise
}
...

```

### ## Explanation

In this corrected version, the expression `n % 2 === 0` evaluates to `true` if `n` is even (i.e., the remainder of `n` divided by 2 is 0) and `false` otherwise.

### ## Example Use Cases

Here are some examples of using the corrected `isEven` function:

```
```javascript
console.log(isEven(10)); // Output: true
console.log(isEven(11)); // Output: false
console.log(isEven(0)); // Output: true
console.log(isEven(-2)); // Output: true
console.log(isEven(-3)); // Output: false
...

```

Note that this function assumes that the input `n` is an integer. If `n` can be a non-integer, you may want to add additional error checking or rounding logic to handle those cases.

Response time: 10.31 seconds

Few-shot Prompt:

Identify and fix the bug:

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

```

Example 1:

The bug is that the function returns `n % 2` which gives 0 for even numbers and 1 for odd numbers, but JavaScript interprets 0 as false and 1 as true. This means `isEven(4)` returns 0 (falsy) when it should return true. The fix is to compare the result to 0:

```
function isEven(n) {
  return n % 2 === 0; // Correctly returns true for even numbers
}

```

Example 2:

The function currently returns the remainder after division by 2 (0 or 1), not a boolean. In JavaScript, 0 is falsy and 1 is truthy, so `isEven(2)` would return 0 which evaluates to false, contradicting the function name. Fix by explicitly checking if the remainder is 0:

```
function isEven(n) {
  return n % 2 === 0; // Now properly returns true for even numbers
}

```

GPT-4o Response:

The examples you provided accurately identify and explain the bug in the `isEven` function and how to fix it. Here's a concise summary:

Bug:

The original function:

```
```javascript
function isEven(n) {
 return n % 2; // Returns 0 for even numbers and 1 for odd numbers, not true/false
}
...

```

returns the remainder of `n % 2`, which is `0` for even numbers and `1` for odd numbers. However, in JavaScript:

- `0` is **falsy**, so the function appears to return `false` for even numbers.
- `1` is **truthy**, so the function appears to return `true` for odd numbers.

This behavior is inconsistent with the function's name `isEven`, which implies it should return a boolean (`true` for even numbers, `false` for odd numbers).

#### ### Fix:

To ensure the function returns a boolean (`true` or `false`), we should explicitly check if the remainder is `0`:

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

```

Explanation:

- `n % 2 === 0` evaluates to `true` if the remainder is `0` (i.e., `n` is even).
- Otherwise, it evaluates to `false` (i.e., `n` is odd).

Examples of Correct Behavior:

```
```javascript
isEven(4); // true (4 is even)
isEven(7); // false (7 is odd)
isEven(0); // true (0 is even)
isEven(-2); // true (-2 is even)
isEven(-3); // false (-3 is odd)
...

```

This fix ensures that the function's behavior aligns with its name and purpose.

Response time: 4.90 seconds

Llama-3.1 Response:

#### The Bug:

The function `isEven(n)` is supposed to return `true` if the number `n` is even and `false` otherwise. However, it currently returns the remainder of `n` divided by 2, which is either 0 or 1. In JavaScript, 0 is considered falsy and 1 is considered truthy. This means that the function is returning incorrect results.

The Fix:

To fix this bug, we need to modify the function to return a boolean value indicating whether the number is even or not. We can do this by comparing the remainder of `n` divided by 2 to 0.

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

Explanation:

In this corrected version, the function uses the modulo operator (`%`) to calculate the remainder of `n` divided by 2. If the remainder is 0, then `n` is even, and the function returns `true`. Otherwise, `n` is odd, and the function returns `false`.

Example Use Cases:

```
```javascript
console.log(isEven(4)); // true
console.log(isEven(3)); // false
console.log(isEven(10)); // true
console.log(isEven(7)); // false
```
```

By making this simple change, the function now correctly returns a boolean value indicating whether the input number is even or odd.

Response time: 9.91 seconds

Prompt & Responses given in lines 4259-4431 of model\_comparison\_results.txt

**Analysis:** Both suggested renaming the function to isEven or change the code from `x % 2 == 0` to `x % 2 != 0`. Few-Prompt did little to change the output for either model.

**Comparison:** No appreciable difference.

## Task 19: Summary Decomposition (C++)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

Here is the decomposition of the high-level comment into logical steps:

1. **\*\*Input Validation\*\***:
  - Check if the input value `x` is less than 0.
  - If the input is invalid (i.e.,  $x < 0$ ), return -1 as an error code.
2. **\*\*Perform Calculation\*\***:
  - If the input is valid (i.e.,  $x \geq 0$ ), calculate the square of the input by multiplying `x` by itself ( $x * x$ ).
3. **\*\*Return the Result\*\***:
  - Return the result of the squared calculation to the caller. If the input was invalid, return -1 instead.

Response time: 2.84 seconds

Llama-3.1 Response:

Here's the decomposition of the high-level comment into logical steps:

1. **\*\*Validate the input\*\***: Check if the input `x` is valid (in this case, non-negative).
  - If `x` is invalid (negative), return an error code (-1).
2. **\*\*Calculate the square\*\***: If `x` is valid, calculate its square ( $x * x$ ).
3. **\*\*Return the result\*\***: Return the calculated square (or the error code if the input was invalid).

Here's the updated code with comments for each step:

```
```c
// Function that validates an input, calculates square, and returns result
int process(int x) {
    // Step 1: Validate the input
    if (x < 0) {
        // Return an error code if input is invalid
        return -1;
    }

    // Step 2: Calculate the square
    int square = x * x;

    // Step 3: Return the result
    return square;
}
```
```

This decomposition makes the code more readable and maintainable, as each step is clearly separated and commented. However, for performance-critical code, the original version may be preferred as it has fewer operations.

Response time: 8.00 seconds

Few-shot Prompt:

```
Decompose the high-level comment/summary into logical steps:
// Function that validates an input, calculates square, and returns result
```

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

Example 1:

```
// 1. Validate that input x is non-negative
// 2. Calculate the square of input x
// 3. Return the calculated result
```

```
int process(int x) {
 // 1. Validate that input x is non-negative
 if (x < 0) return -1;
 // 2. Calculate the square of input x
 // 3. Return the calculated result
 return x * x;
}
```

Example 2:

```
/**
 * Process a number by:
 * 1. Validating that the input is non-negative
 * 2. Computing the square of the input
 * 3. Returning the computed result or error code
 */
int process(int x) {
 // Validate input is non-negative
 if (x < 0) return -1; // Return error code for invalid input

 // Calculate and return square
 return x * x;
}
```

GPT-4o Response:

The high-level comment summarizing the function can be decomposed into the following logical steps:

1. **Input Validation**:
  - Check if the input `x` is non-negative.
  - If the input is negative, return an error code (`-1` in this case).
2. **Computation**:
  - If the input passes validation, calculate the square of the input (`x \* x`).
3. **Result Handling**:
  - Return the calculated square if the input is valid.
  - Otherwise, return the error code (`-1`).

These steps represent the typical flow of the function:

1. Input validation ensures the function only operates on valid data.
2. The computation step performs the necessary processing (squaring the input).
3. Result handling ensures the appropriate value is returned to the caller, whether it's a valid result or an error code.

Response time: 2.54 seconds

Llama-3.1 Response:

Here's the decomposition of the high-level comment/summary into logical steps:

1. **\*\*Validate Input\*\***: Ensure that the input `x` is non-negative.
  - Check if `x` is less than 0.
  - If `x` is negative, return an error code (in this case, -1).
2. **\*\*Calculate Square\*\***: Compute the square of the input `x`.
  - Use the formula  $x * x$  to calculate the square.
3. **\*\*Return Result\*\***: Return the calculated result or an error code if the input is invalid.
  - If the input is valid, return the calculated square.
  - If the input is invalid (negative), return the error code (-1).

By breaking down the process into these logical steps, you can clearly understand the flow of the function and implement it accordingly. Here's the code with these steps applied:

```
```cpp
/**
 * Process a number by:
 * 1. Validating that the input is non-negative
 * 2. Computing the square of the input
 * 3. Returning the computed result or error code
 */
int process(int x) {
    // Step 1: Validate input is non-negative
    if (x < 0) {
        // Return error code for invalid input
        return -1;
    }

    // Step 2: Calculate square
    int square = x * x;

    // Step 3: Return the calculated result
    return square;
}
```
```

This code follows the same logic as the original examples but with a clearer separation of concerns and more explicit comments.

Response time: 10.72 seconds

Prompt & Responses given in lines 4432-4600 of model\_comparison\_results.txt

**Analysis:** Both decomposed the code well. GPT-4o used a structured, numbered breakdown, while Llama likewise used a numbered breakdown; it also included a rewritten example allowing for as much information in each step. It also points out that this new code is less efficient. Few-Prompt did little to change the responses.

**Comparison:** Use either GPT-4o or Llama. For learning use Llama for flow understanding.

## Task 20: Purpose Inference -> Completion (Python)

Prompting Strategy 1: Zero-shot Prompt:

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

GPT-4o Response:

```
Here's how you can complete the function to calculate the average of the `scores` list:

```python
def calculate_average(scores):
    total = sum(scores) # Calculate the sum of all scores
    count = len(scores) # Count the number of scores
    if count == 0: # Avoid division by zero
        return 0
    return total / count # Calculate and return the average
```

Explanation:
1. `sum(scores)` calculates the total of all elements in the list `scores`.
2. `len(scores)` gives the number of elements in the list, which is the count.
3. To avoid dividing by zero (in case the input list is empty), a check `if count == 0` is added. If the list is empty, the function returns `0`.
4. Otherwise, the average is calculated using `total / count` and returned.

Example Usage:
```python
scores = [80, 90, 70, 85]
print(calculate_average(scores)) # Output: 81.25

empty_scores = []
print(calculate_average(empty_scores)) # Output: 0
```
```

Response time: 5.07 seconds

Llama-3.1 Response:

```
```python
def calculate_average(scores):
    """
    Calculate the average of a list of scores.

    Args:
        scores (list): A list of numeric scores.

    Returns:
        float: The average of the scores.

    Raises:
        ValueError: If the input list is empty.
        TypeError: If the input list contains non-numeric values.
    """
    if not scores:
        raise ValueError("Input list cannot be empty")

    total = 0
    for score in scores:
        if not isinstance(score, (int, float)):
            raise TypeError("Input list must only contain numeric values")
        total += score

    average = total / len(scores)
    return average
```
```

Example use case:

```
```python
scores = [90, 80, 70, 60]
average = calculate_average(scores)
print(f"The average score is: {average}")
```
```

Alternatively, you can use the built-in `sum()` function and a list comprehension to simplify the function:

```
```python
def calculate_average(scores):
    if not scores:
        raise ValueError("Input list cannot be empty")

    if not all(isinstance(score, (int, float)) for score in scores):
        raise TypeError("Input list must only contain numeric values")

    average = sum(scores) / len(scores)
    return average
```
```

Response time: 69.74 seconds

#### Few-shot Prompt:

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

Example 1:
def calculate_average(scores):
 total = 0
 for score in scores:
 total += score
 return total / len(scores) if scores else 0

Example 2:
def calculate_average(scores):
 total = 0
 if not scores:
 return 0 # Handle empty list case

 for score in scores:
 total += score

 # Calculate and return the average
 return total / len(scores)
```

#### GPT-4o Response:

Here is the complete function based on the provided intent and examples:

```
```python
def calculate_average(scores):
    total = 0
    if not scores: # Check if the list is empty
        return 0 # Return 0 if the list is empty

    for score in scores: # Iterate through each score in the list
        total += score # Add each score to the total

    return total / len(scores) # Calculate and return the average
```
```

#### ### Explanation:

1. **Handling the empty list case**: The function checks if the `scores` list is empty. If it is, it returns `0` immediately to avoid dividing by `0`.



2. **\*\*Summing the scores\*\***: A `for` loop is used to iterate through the scores in the list and accumulate their sum in the `total` variable.

3. **\*\*Calculating the average\*\***: Once the loop finishes, the total sum is divided by the number of scores (`len(scores)`) to calculate the average, which is then returned.

### Example Usage:

```
```python
```

```
# Example 1:
```

```
scores = [80, 90, 100]
```

```
print(calculate_average(scores)) # Output: 90.0
```

```
# Example 2:
```

```
scores = []
```

```
print(calculate_average(scores)) # Output: 0
```

```
# Example 3:
```

```
scores = [70, 85]
```

```
print(calculate_average(scores)) # Output: 77.5
```

```
```
```

This function is now complete and handles both non-empty and empty lists of scores correctly.

Response time: 4.60 seconds

Llama-3.1 Response:

Here's a completed and well-structured version of the function:

```
```python
```

```
def calculate_average(scores):
```

```
    """
```

```
    Calculate the average of a list of scores.
```

```
    Args:
```

```
    scores (list): A list of scores.
```

```
    Returns:
```

```
    float: The average score. Returns 0 if the list is empty.
```

```
    """
```

```
    if not scores:
```

```
        return 0 # Handle empty list case
```

```
    # Calculate the sum of scores using the built-in sum function
```

```
    total = sum(scores)
```

```
    # Calculate and return the average
```

```
    return total / len(scores)
```

```
```
```

Alternatively, you can use a more concise version using the `statistics` module:

```
```python
import statistics

def calculate_average(scores):
    """
    Calculate the average of a list of scores.

    Args:
        scores (list): A list of scores.

    Returns:
        float: The average score. Returns 0 if the list is empty.
    """
    return statistics.mean(scores) if scores else 0
```
```

Or without importing any modules:

```
```python
def calculate_average(scores):
    """
    Calculate the average of a list of scores.

    Args:
        scores (list): A list of scores.

    Returns:
        float: The average score. Returns 0 if the list is empty.
    """
    return sum(scores) / len(scores) if scores else 0
```
```

Response time: 9.13 seconds

**Analysis:** Both completed the average calculation correctly. GPT-4o's solution was precise, while Llama added error handling (e.g., empty list checks, checked for non-numbers). With Few-Prompt GPT-4o stopped using the python function sum and instead inserted a for loop. Llama continued to use the sum function.

**Comparison:** Select Llama for robust code and detailed comments.

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

Prompting Strategy 1: Zero-shot Prompt:

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

```
utils.py - Script to parse and summarize numeric CSV files
import csv
def read_csv(filepath):
 with open(filepath, 'r') as f:
 return [row for row in csv.reader(f)]
def summarize_column(data, index):
 values = [float(row[index]) for row in data[1:]] # skip header
 total = sum(values)
 avg = total / len(values)
 return total, avg
def main():
 filepath = 'data.csv'
 data = read_csv(filepath)
 total, avg = summarize_column(data, 1)
 print("Total:", total)
 print("Average:", avg)
if __name__ == '__main__':
 main()
```

GPT-4o Response:

#### Analysis of the Script

##### Issues and Flaws in the Script:

1. **\*\*Hardcoded File Path\*\***:

- The file path `data.csv` is hardcoded in the `main()` function. This makes the script less reusable and inflexible.

2. **\*\*No Error Handling\*\***:

- There is no error handling for potential issues, such as:
  - File not found (`FileNotFoundError`).
  - Invalid file format (e.g., non-CSV files).
  - Index out of range or missing data.
  - Non-numeric data in the target column.
- Lack of error handling makes the script brittle and prone to crashing.

3. **\*\*Assumes Header Row Exists\*\***:

- The script assumes the first row is a header and skips it, which may not always be the case.

4. **\*\*No Input Validation\*\***:

- The script does not validate the `index` parameter passed to `summarize\_column`, leading to potential `IndexError` if the column index is invalid.

5. **\*\*Readability Issues\*\***:

- The code lacks comments and could be more readable with better variable names and structure.
- The function names (`read\_csv`, `summarize\_column`) could be more descriptive to indicate their purpose clearly.

6. **\*\*Global Dependencies\*\***:

- The `filepath` variable is hardcoded in `main()` rather than being passed as a parameter or argument. This design is not conducive for testing or reuse.

7. **\*\*Unnecessary List Comprehension\*\***:

- The `read\_csv` function uses a list comprehension to load the entire file into memory. This is unnecessary for large files where streaming might suffice.

8. **\*\*Division by Zero\*\***:

- The `summarize\_column` function does not check if the list of values is empty, leading to a potential `ZeroDivisionError`.

---

### Refactored Script

Below is a refactored version of the script addressing the issues mentioned above:

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

def read_csv(filepath):
    """
    Reads a CSV file and returns its content as a list of rows.
    Each row is represented as a list of strings.
    """
    try:
        with open(filepath, 'r') as file:
            reader = csv.reader(file)
            return [row for row in reader]
    except FileNotFoundError:
        raise FileNotFoundError(f"File not found: {filepath}")
    except Exception as e:
        raise RuntimeError(f"An error occurred while reading the file: {e}")

def summarize_column(data, index):
    """
    Summarizes a numeric column from the given CSV data.
    Skips the header row and calculates the total and average.

    Args:
        data (list of list of str): The CSV data as a list of rows.
        index (int): The column index to summarize.

    Returns:
        tuple: A tuple containing the total and average of the column.
    """
```

Raises:

ValueError: If the column contains non-numeric data or is empty.

IndexError: If the index is out of range.

"""

try:

Ensure there is data beyond the header

if len(data) <= 1:

raise ValueError("No data rows found in the CSV file.")

Extract numeric values from the specified column

values = []

for row in data[1:]: # Skip the header

try:

values.append(float(row[index]))

except (ValueError, IndexError):

raise ValueError(f"Non-numeric or missing data in column {index}.")

if not values:

raise ValueError("The specified column is empty.")

total = sum(values)

avg = total / len(values)

return total, avg

except IndexError:

raise IndexError(f"Column index {index} is out of range.")

except Exception as e:

raise RuntimeError(f"An error occurred while summarizing the column: {e}")

def main(filepath, column_index):

"""

Main function to read a CSV file, summarize a specific column, and print the results.

Args:

filepath (str): The path to the CSV file.

column_index (int): The index of the column to summarize.

"""

try:

data = read_csv(filepath)

total, avg = summarize_column(data, column_index)

print(f"Total: {total}")

print(f"Average: {avg}")

except Exception as e:

print(f"Error: {e}")

if __name__ == '__main__':

import argparse

Set up argument parsing for a more flexible script

```
parser = argparse.ArgumentParser(description="Summarize a numeric column in a CSV file.")
parser.add_argument('filepath', type=str, help="Path to the CSV file.")
parser.add_argument('column_index', type=int, help="
```

Response time: 28.37 seconds

Llama-3.1 Response:

****Analysis of the Script****

The given utility script is designed to read a CSV file, parse its contents, and calculate the total and average of a specific column. Here are some potential logic issues and design flaws:

1. ****Error Handling****: The script does not handle potential errors that may occur when reading the CSV file, such as file not found, permission issues, or malformed CSV data. It assumes that the file exists, is readable, and has a valid CSV format.
2. ****Data Validation****: The script does not validate the data before attempting to convert it to a float. If the column contains non-numeric data, a `ValueError` will be raised.
3. ****Column Index****: The script hardcodes the column index to 1, which may not be the desired column in all cases.
4. ****CSV Format****: The script assumes that the CSV file has a header row. If the file does not have a header row, the script will incorrectly skip the first row of data.
5. ****Division by Zero****: If the column is empty, the script will raise a `ZeroDivisionError` when calculating the average.

****Refactored Code****

Here's a refactored version of the script that addresses the above issues:

```
```python
import csv

def read_csv(filepath):
 """
 Reads a CSV file and returns its contents as a list of lists.

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

 Returns:
 list: A list of lists, where each inner list represents a row in the CSV file.

 Raises:
 FileNotFoundError: If the file does not exist.
 PermissionError: If the file is not readable.
 csv.Error: If the file is not a valid CSV file.
 """
 try:
 with open(filepath, 'r') as f:
```

```

 return [row for row in csv.reader(f)]
 except FileNotFoundError:
 print(f"Error: File '{filepath}' not found.")
 return []
 except PermissionError:
 print(f"Error: Permission denied to read file '{filepath}'.")
 return []
 except csv.Error as e:
 print(f"Error: Failed to parse CSV file '{filepath}'. {e}")
 return []

def summarize_column(data, index, skip_header=True):
 """
 Calculates the total and average of a specific column in the given data.

 Args:
 data (list): A list of lists, where each inner list represents a row in the data.
 index (int): The index of the column to summarize.
 skip_header (bool): Whether to skip the first row (header) when summarizing. Defaults to True.

 Returns:
 tuple: A tuple containing the total and average of the specified column.

 Raises:
 ValueError: If the column contains non-numeric data.
 IndexError: If the column index is out of range.
 """
 if not data:
 return 0, 0

 if skip_header:
 data = data[1:]

 try:
 values = [float(row[index]) for row in data]
 total = sum(values)
 avg = total / len(values) if values else 0
 return total, avg
 except ValueError:
 print("Error: Column contains non-numeric data.")
 return 0, 0
 except IndexError:
 print("Error: Column index out of range.")
 return 0, 0

def main():
 filepath = 'data.csv'
 column_index = 1

```

```

data = read_csv(filepath)
if data:
 total, avg = summarize_column(data, column_index)
 print(f"Total of column {column_index + 1}: {total}")
 print(f"Average of column {column_index + 1}: {avg}")

if __name__ == '__main__':
 main()
'''
Changes Made

```

1. **\*\*Added Error Handling\*\***: The `read\_csv` function now handles potential errors when reading the CSV file, and the `summarize\_column` function handles errors when converting data to float.
2. **\*\*Added Data Validation\*\***: The `summarize\_column` function checks if the column contains non-numeric data and raises a ValueError if it does.
3. **\*\*Made Column Index Configurable\*\***: The column index is now a parameter to the `summarize\_column` function, making it easier to summarize different columns.
4. **\*\*Added Option to Skip Header\*\***: The `summarize\_column` function now has an optional `skip\_header` parameter, which allows the user to choose whether to skip the first row (header) when summarizing.
5. **\*\*Improved Code Readability\*\***: The code has been refactored to improve readability, with clearer function names, docstrings, and error messages.

Response time: \_\_\_\_ seconds

### Few-shot Prompt:

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

```

utils.py - Script to parse and summarize numeric CSV files
import csv
def read_csv(filepath):
 with open(filepath, 'r') as f:
 return [row for row in csv.reader(f)]
def summarize_column(data, index):
 values = [float(row[index]) for row in data[1:]] # skip header
 total = sum(values)
 avg = total / len(values)
 return total, avg
def main():
 filepath = 'data.csv'
 data = read_csv(filepath)
 total, avg = summarize_column(data, 1)
 print("Total:", total)
 print("Average:", avg)
if __name__ == '__main__':
 main()

```

Example 1:



I've identified several issues with this script:

1. Error handling: The script doesn't handle file not found errors, invalid data, or empty columns.
2. Hardcoded values: The filepath and column index are hardcoded rather than configurable.
3. Type safety: There's no validation that the data contains numeric values.
4. Documentation: The code lacks docstrings and comments.

Here's my refactored version:

```
```python
# utils.py - Script to parse and summarize numeric CSV files
import csv
import sys
from typing import List, Tuple, Any

def read_csv(filepath: str) -> List[List[str]]:
    """Read a CSV file and return its contents as a list of rows.

    Args:
        filepath: Path to the CSV file

    Returns:
        List of rows, where each row is a list of strings

    Raises:
        FileNotFoundError: If the file doesn't exist
    """
    try:
        with open(filepath, 'r') as f:
            return list(csv.reader(f))
    except FileNotFoundError:
        print(f"Error: File {filepath} not found.")
        raise

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

    Args:
        data: List of rows from a CSV file
        index: Index of the column to summarize (0-based)

    Returns:
        Tuple containing (total, average) for the column

    Raises:
        IndexError: If the column index is out of range
        ValueError: If the column contains non-numeric values
        ZeroDivisionError: If the data has no rows (excluding header)
    """
```

```

if not data or len(data) <= 1:
    raise ValueError("Data must have at least one row plus a header")

if index < 0 or (data and index >= len(data[0])):
    raise IndexError(f"Column index {index} is out of range")

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

    total = sum(values)
    avg = total / len(values)
    return total, avg
except ValueError as e:
    if "could not convert string to float" in str(e):
        print(f"Error: Column {index} contains non-numeric values")
    raise

def main(filepath: str = None, column_index: int = None):
    """Parse and summarize a column from a CSV file.

    Args:
        filepath: Path to the CSV file, defaults to 'data.csv'
        column_index: Index of the column to summarize (0-based), defaults to 1
    """
    # Use command line arguments or defaults
    filepath = filepath or (sys.argv[1] if len(sys.argv) > 1 else 'data.csv')
    column_index = column_index or (int(sys.argv[2]) if len(sys.argv) > 2 else 1)

    try:
        data = read_csv(filepath)
        total, avg = summarize_column(data, column_index)
        print(f"Column: {data[0][column_index] if data and data[0] else column_index}")
        print(f"Total: {total:.2f}")
        print(f"Average: {avg:.2f}")
    except (FileNotFoundError, IndexError, ValueError, ZeroDivisionError) as e:
        print(f"Error: {e}")
        sys.exit(1)

if __name__ == '__main__':
    main()

```

This refactored version includes:

- Proper error handling for file operations and data processing
- Command-line argument support with sensible defaults
- Type hints for better code understanding
- Comprehensive docstrings explaining function purposes

- Safer numeric conversion with error handling
- Formatted output for better readability

Example 2:

I've analyzed the utility script and found several issues:

1. No error handling - The script doesn't handle missing files, invalid data formats, or edge cases
2. Hardcoded filepath - 'data.csv' is hardcoded in main(), making the script inflexible
3. No validation - The script assumes column 1 exists and contains numeric data
4. Division by zero risk - If data has only a header row, len(values) will be 0
5. No proper documentation or type hints

Here's my refactored version:

```
```python
#!/usr/bin/env python3
"""Utility script to parse and summarize numeric data from CSV files."""

import csv
import os
import sys
from typing import List, Tuple, Optional

def read_csv(filepath: str) -> List[List[str]]:
 """Read data from a CSV file.

 Args:
 filepath: Path to the CSV file

 Returns:
 A list of rows, where each row is a list of strings

 Raises:
 FileNotFoundError: If the file doesn't exist
 """
 if not os.path.exists(filepath):
 raise FileNotFoundError(f"File not found: {filepath}")

 with open(filepath, 'r', newline=") as f:
 return list(csv.reader(f))

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

 Args:
 data: CSV data as a list of rows
 index: Column index to summarize (0-based)
 """
```

Returns:

A tuple containing (total, average)

Raises:

IndexError: If the column index is out of bounds

ValueError: If the column contains non-numeric values

ValueError: If there's no data to average (empty dataset)

"""

if not data:

raise ValueError("Empty dataset provided")

if len(data) <= 1:

raise ValueError("Dataset contains only headers, no values to summarize")

# Validate column index

if index < 0 or index >= len(data[0]):

raise IndexError(f"Column index {index} is out of bounds for data with {len(data[0])} columns")

try:

# Convert values to float, skipping the header row

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

if not values:

raise ValueError(f"No valid numeric values found in column {index}")

total = sum(values)

avg = total / len(values)

return total, avg

except ValueError as e:

if "could not convert string to float" in str(e):

raise ValueError(f"Column {index} contains non-numeric values")

raise

def parse\_args() -> Tuple[str, int]:

"""Parse command line arguments or use defaults.

Returns:

Tuple of (filepath, column\_index)

"""

filepath = 'data.csv' # Default filepath

column\_index = 1 # Default column index

if len(sys.argv) > 1:

filepath = sys.argv[1]

if len(sys.argv) > 2:

try:

column\_index = int(sys.argv[2])

except ValueError:

print(f"Error: Column index must be an integer, got '{sys.argv[2]}")

```

 sys.exit(1)

 return filepath, column_index

def main() -> None:
 """Main function to process CSV and display column statistics."""
 filepath, column_index = parse_args()

 try:
 print(f"Processing file: {filepath}, column: {column_index}")
 data = read_csv(filepath)

 # Get column name from header if available
 column_name = data[0][column_index] if data and len(data[0]) > column_index else f"Column {column_index}"

 total, avg = summarize_column(data, column_index)

 print(f"Column: {column_name}")
 print(f"Total: {total:.2f}")
 print(f"Average: {avg:.2f}")
 print(f"Row count: {len(data) - 1}")
 except (FileNotFoundError, IndexError, ValueError) as e:
 print(f"Error: {e}")
 sys.exit(1)

if __name__ == '__main__':
 main()

```

Improvements in this refactored version:

1. Added comprehensive error handling throughout the code
2. Added proper type hints and docstrings
3. Implemented command-line argument parsing
4. Added validation for inputs and data formats
5. Enhanced output with formatted values and additional information
6. Made the code more robust against edge cases like empty files or non-numeric data
7. Used proper CSV handling with `newline=""` parameter
8. Added structure with clear function responsibilities

GPT-4o Response:

Both refactored versions provide significant improvements over the original script. However, I'll summarize the key enhancements and point out a few additional considerations to make the code even better.

---

### Key Enhancements in the Refactored Versions:

1. **Error Handling**:

- Both versions improved error handling for file operations, invalid column indices, and non-numeric data.

- They handle edge cases like missing files, empty datasets, and non-numeric values gracefully.

## 2. **Flexibility**:

- Both refactored versions allow users to specify the file path and column index via command-line arguments, improving the script's reusability.

- Default values (`data.csv` for the file and `1` for the column index) are provided for convenience.

## 3. **Type Safety and Validation**:

- Added type hints for function arguments and return values to improve code readability and safety.

- Validated column indices and ensured numeric conversion of data.

## 4. **Documentation**:

- Added concise and clear docstrings for functions and the overall script.

- Explained error scenarios and return values, making the code easier to understand and maintain.

## 5. **Output Formatting**:

- Both versions format the outputs (e.g., `Total: 123.45`, `Average: 67.89`) for better readability.

- Included additional information like the column name and row count.

---

## ### Additional Opportunities for Further Improvement:

While both refactored versions are robust, here are a few more improvements to make the script even more comprehensive and user-friendly:

### 1. **Improved Argument Parsing**:

- Use a library like `argparse` for parsing command-line arguments. This provides better handling of optional arguments, default values, and help messages.

- Example:

```
```python
import argparse
```

```
def parse_args() -> Tuple[str, int]:
```

```
    parser = argparse.ArgumentParser(description="Summarize numeric data from a CSV file.")
```

```
    parser.add_argument("filepath", type=str, help="Path to the CSV file")
```

```
    parser.add_argument("column_index", type=int, help="Index of the column to summarize (0-based)")
```

```
    return parser.parse_args()
```

```
```
```

### 2. **CSV Dialect Handling**:

- Consider using `csv.Sniffer` to auto-detect the CSV file format (e.g., delimiter, quote character) for better compatibility with non-standard CSV files.

- Example:

```
```python
```

```
with open(filepath, 'r', newline='') as f:
```

```
    dialect = csv.Sniffer().sniff(f.read(1024))
```

```
    f.seek(0)
```

```
    reader = csv.reader(f, dialect)
```

```
    return list(reader)
```

```

...
3. **Handling Missing or Empty Cells**:
- Skip rows with empty cells in the target column or provide an option to treat them as zero.
- Example:
    ```python
 values = [float(row[index]) for row in data[1:] if row[index].strip()]
    ```
4. **Support for Multiple Columns**:
- Allow summarizing multiple columns at once by accepting a list of column indices.
- Example:
    ```python
 def summarize_columns(data: List[List[str]], indices: List[int]) -> List[Tuple[float, float]]:
 results = []
 for index in indices:
 results.append(summarize_column(data, index))
 return results
    ```
5. **Unit Testing**:
- Add unit tests to validate the functionality of individual functions.
- Example framework: `unittest` or `pytest`.
6. **Logging**:
- Use the `logging` module instead of `print` for error messages and debugging output. This allows for better control of log levels and output destinations.
- Example:
    ```python
 import logging
 logging.basicConfig(level=logging.INFO)
 logging.error("File not found: %s", filepath)
    ```
7. **Optional Output Formats**:
- Allow users to export the summary (e.g., total and average) to a file in CSV or JSON format for better usability in automation scripts.
---
### Final Refactored Script:
Here's the updated version incorporating the above suggestions:

    ```python
 #!/usr/bin/env python3
 """Utility script to parse and summarize numeric data from CSV files."""

 import csv
 import argparse
 import sys
 from typing import List, Tuple, Optional

 def read_csv(filepath: str) -> List[List[str]]:
 """Read data from a CSV file."""

```

```

try:
 with open(filepath, 'r', newline='') as f:
 dialect = csv.Sniffer().sniff(f.read(1024))
 f.seek(0)
 reader = csv.reader(f, dialect)
 return list(reader)
except FileNotFoundError:
 raise FileNotFoundError(f"File not found: {filepath}")
except csv.Error as e:
 raise ValueError(f"Error reading CSV file: {e}")

```

Response time: 11.68 seconds

Llama-3.1 Response:

Both refactored versions of the utility script address the original issues and make significant improvements in terms of error handling, code organization, and documentation. Here's a comparison of the two versions and some additional suggestions for further improvement:

**Similarities:**

1. Both versions add comprehensive error handling for file operations, data processing, and edge cases.
2. They implement command-line argument parsing with sensible defaults.
3. Both versions use proper type hints and docstrings to improve code readability and understanding.
4. They validate inputs and data formats to prevent errors.

**Differences:**

1. **Error handling approach:** The first refactored version uses a more explicit approach to error handling, where each function raises specific exceptions. The second version uses a more general approach, where exceptions are caught and handled at the top level.
2. **Command-line argument parsing:** The first version uses a simpler approach to parsing command-line arguments, while the second version uses a separate function (`parse_args`) to handle argument parsing.
3. **Code organization:** The first version has a more linear structure, while the second version uses separate functions for each task (e.g., `read_csv`, `summarize_column`, `parse_args`).
4. **Output formatting:** The second version includes more detailed output, such as the column name and row count.

**Suggestions for further improvement:**

1. **Consider using a more robust CSV parsing library:** The built-in `csv` library can be limited in its capabilities. Consider using a more advanced library like `pandas` or `csvkit`.
2. **Add support for multiple columns:** Currently, the script only supports summarizing a single column. Consider adding support for multiple columns.
3. **Improve error messages:** While the script handles errors well, the error messages could be more informative. Consider adding more details to the error messages to help users diagnose issues.
4. **Add a `--help` option:** Consider adding a `--help` option to display usage instructions and available options.



5. **\*\*Use a consistent coding style:\*\*** Both versions use a mix of coding styles. Consider using a consistent style throughout the codebase.
6. **\*\*Consider adding unit tests:\*\*** To ensure the script works correctly, consider adding unit tests to verify its functionality.

Overall, both refactored versions are significant improvements over the original script. With some additional tweaks and improvements, the script can become even more robust and user-friendly.

Response time: \_\_\_\_ seconds

**Analysis:** GPT-4o and Llama identified and addressed issues in the original script, such as hardcoded paths, lack of error handling, and potential division-by-zero errors. In Zero-Shot, GPT-4o provided a refactored script with robust error handling, command-line argument support via `argparse`, and documentation. Llama's response also added error handling and a `skip_header` option, but retained a hardcoded filepath and column index. In Few-Shot, GPT-4o summarized the provided improvements (e.g., type hints, command-line support) and proposed a refactored script with `csv.Sniffer` for dialect detection, though it was incomplete (cut off mid-code). Llama's few-shot response analyzed the examples, suggesting enhancements like `pandas` or unit tests, but did not provide a refactored script, missing the core task requirement. Neither model made critical errors, but GPT-4o's incomplete scripts and Llama's lack of a few-shot implementation were notable shortcomings.

**Comparison:** GPT-4o performed better overall, in zero-shot, for delivering a nearly complete, refactoring.

## Task 22: Code Completion and Robustness Enhancement (Python)

Prompting Strategy 1: Zero-shot Prompt:

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

```
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()
```

GPT-4o Response:

Here's the completed `file\_processor.py` script with the missing functionality filled in:

```
```python
# file_processor.py - Complete script for processing text files
import string

def load_file(filepath):
    """
    Reads a file and returns its lines as a list.
    """
    with open(filepath, 'r') as f:
        return f.readlines()

def clean_line(line):
    """
    Removes punctuation from a line, converts it to lowercase, and trims whitespace.
    """
    # Translate punctuation to None and make lowercase
    translator = str.maketrans("", "", string.punctuation)
    return line.translate(translator).lower().strip()

def count_words(lines):
    """
    Counts the frequency of words across all 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():
    """
    Main function that orchestrates the file processing.
    """
    filepath = 'input.txt' # The input file containing text
    lines = load_file(filepath)
```

```

counts = count_words(lines)
for word, count in sorted(counts.items()):
    print(f"{word}: {count}")

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

#### Explanation of Changes:
1. clean_line` Function:
    - Used str.maketrans` and str.translate` to remove all punctuation.
    - Converted the line to lowercase using str.lower()` to ensure case-insensitivity.
    - Used str.strip()` to remove leading/trailing whitespace.

2. count_words` Function:
    - This function was already written correctly; it splits the cleaned line into words and counts their occurrences.

3. Main Function:
    - Reads the file, processes the lines, and prints the word counts sorted alphabetically.

#### Example Usage:
Suppose input.txt` contains the following text:
...
Hello world! This is a test.
Hello, this is another test. Testing, testing, 123.
..

Running the script will produce:
...
123: 1
a: 1
another: 1
hello: 2
is: 2
test: 2
testing: 2
this: 2
world: 1
...

This script ensures proper handling of punctuation and case sensitivity while counting word frequencies.

```

Response time: 16.70 seconds

Llama-3.1 Response:

```

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

def load_file(filepath):

```

```

"""
Reads a file line by line and returns the lines as a list.

Args:
 filepath (str): The path to the file.

Returns:
 list: A list of lines from the file.
"""
with open(filepath, 'r') as f:
 return f.readlines()

def clean_line(line):
 """
 Removes punctuation from a line and converts it to lowercase.

 Args:
 line (str): The line to clean.

 Returns:
 str: The cleaned line.
 """
 # Remove leading/trailing whitespace
 line = line.strip()
 # Remove punctuation
 line = line.translate(str.maketrans("", "", string.punctuation))
 # Make lowercase
 line = line.lower()
 return line

def count_words(lines):
 """
 Counts the frequency of words in a list of lines.

 Args:
 lines (list): The list of lines to count words from.

 Returns:
 dict: A dictionary of word frequencies.
 """
 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():
 """

```

The main entry point of the script.

```
"""
```

```
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()
'''
```

This script will correctly read a text file, clean each line by removing punctuation and converting to lowercase, and then count the frequency of each word. The result will be printed in alphabetical order.

Note that the `clean\_line` function now uses the `translate` method to remove punctuation, which is more efficient than using regular expressions or iteration. The `strip` method is used to remove leading/trailing whitespace, and the `lower` method is used to convert the line to lowercase.

Example use case:

Suppose you have a text file `input.txt` containing the following text:

```
'''
```

```
Hello world!
This is a test.
Hello again!
'''
```

If you run the script, it will output:

```
'''
```

```
a: 1
again: 1
hello: 2
is: 1
test: 1
this: 1
world: 1
'''
```

Response time: 19.44 seconds

Few-shot Prompt:

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

```
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()

```

Example 1:

I'll complete the missing `clean\_line` function and enhance the script for better robustness. Here's the completed version:

```

'''python
file_processor.py - Script for processing text files and counting word frequencies
import string
import sys

def load_file(filepath):
 """Load a text file and return its content as a list of lines.

 Args:
 filepath: Path to the text file

 Returns:
 List of strings, each representing a line in the file

 Raises:
 FileNotFoundError: If the file doesn't exist
 """
 try:
 with open(filepath, 'r', encoding='utf-8') as f:
 return f.readlines()
 except FileNotFoundError:
 print(f"Error: File '{filepath}' not found.")
 sys.exit(1)
 except Exception as e:
 print(f"Error reading file: {e}")

```

```

 sys.exit(1)

def clean_line(line):
 """Clean a line by removing punctuation and converting to lowercase.

 Args:
 line: A string representing a line of text

 Returns:
 Cleaned string with punctuation removed and all lowercase
 """
 # Remove punctuation and make lowercase
 translator = str.maketrans("", "", string.punctuation)
 return line.translate(translator).lower().strip()

def count_words(lines):
 """Count frequency of each word in the provided lines.

 Args:
 lines: List of strings, each representing a line of text

 Returns:
 Dictionary mapping words to their frequency counts
 """
 word_counts = {}
 for line in lines:
 clean = clean_line(line)
 for word in clean.split():
 if word: # Skip empty strings
 word_counts[word] = word_counts.get(word, 0) + 1
 return word_counts

def main():
 """Main function to process text file and display word counts."""
 # Get filepath from command line arguments or use default
 filepath = sys.argv[1] if len(sys.argv) > 1 else 'input.txt'

 print(f"Processing file: {filepath}")
 lines = load_file(filepath)

 counts = count_words(lines)

 if not counts:
 print("No words found in the file.")
 return

 print(f"Found {len(counts)} unique words:")
 for word, count in sorted(counts.items()):
 print(f"{word}: {count}")

```

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

Key improvements:

1. Implemented the `clean\_line` function to remove punctuation and convert text to lowercase
2. Added error handling for file operations
3. Added support for command-line arguments to specify the input file
4. Added skipping of empty strings after cleaning and splitting
5. Added informative output messages
6. Added docstrings for better code documentation
7. Used UTF-8 encoding to support international characters

Example 2:

I've completed the script by implementing the `clean\_line` function and improving the code's robustness. Here's the enhanced version:

```
```python  
#!/usr/bin/env python3  
"""File processor script that cleans text and counts word frequencies."""
```

```
import string  
import sys  
import os  
from collections import Counter
```

```
def load_file(filepath):  
    """Load a text file and return its content as a list of lines.
```

Args:

filepath: Path to the text file to process

Returns:

List of strings where each string is a line from the file

Raises:

FileNotFoundError: If the specified file doesn't exist

```
"""
```

```
if not os.path.exists(filepath):  
    raise FileNotFoundError(f"File not found: {filepath}")
```

```
try:
```

```
    with open(filepath, 'r', encoding='utf-8') as f:  
        return f.readlines()
```

```
except UnicodeDecodeError:
```

```
    # Try again with a different encoding if UTF-8 fails
```

```
    with open(filepath, 'r', encoding='latin-1') as f:  
        return f.readlines()
```



```
def clean_line(line):
    """Remove punctuation from a line and convert to lowercase.

    Args:
        line: String representing a line of text

    Returns:
        Cleaned string with no punctuation and all lowercase characters
    """
    # Create a translation table to remove all punctuation
    translator = str.maketrans("", "", string.punctuation)

    # Remove punctuation, convert to lowercase, and strip whitespace
    cleaned = line.translate(translator).lower().strip()

    # Replace multiple spaces with a single space
    cleaned = ' '.join(cleaned.split())

    return cleaned
```

```
def count_words(lines):
    """Count the frequency of each word in the given lines.

    Args:
        lines: List of text lines to process

    Returns:
        Dictionary mapping words to their frequency counts
    """
    word_counts = {}
    for line in lines:
        if not line.strip(): # Skip empty lines
            continue

        clean = clean_line(line)
        for word in clean.split():
            if word: # Skip empty strings
                word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts
```

```
def parse_args():
    """Parse command line arguments.

    Returns:
        String representing the filepath to process
    """
    if len(sys.argv) > 1:
        return sys.argv[1]
    return 'input.txt' # Default filepath
```

```

def main():
    """Main function to process a text file and print word frequencies."""
    try:
        filepath = parse_args()
        print(f"Processing file: {filepath}")

        lines = load_file(filepath)
        print(f"Loaded {len(lines)} lines")

        counts = count_words(lines)
        print(f"Found {len(counts)} unique words")

        if not counts:
            print("No words found in the file.")
            return

        # Get the top 10 most common words
        most_common = Counter(counts).most_common(10)
        print("
Top 10 most common words:")
        for word, count in most_common:
            print(f"{word}: {count}")

        print("
All words in alphabetical order:")
        for word, count in sorted(counts.items()):
            print(f"{word}: {count}")

        except FileNotFoundError as e:
            print(f"Error: {e}")
            sys.exit(1)
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
            sys.exit(1)

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

```

Improvements made:

1. Implemented the `clean_line` function to remove punctuation and convert text to lowercase
2. Added comprehensive error handling throughout the code
3. Added support for different file encodings
4. Added filtering to skip empty lines and words
5. Included command-line argument support
6. Used the Counter class to efficiently find the most common words
7. Added detailed output information including stats about processed lines and unique words
8. Added proper docstrings and comments for better code documentation
9. Normalized whitespace by replacing multiple spaces with a single space

10. Added graceful error handling with informative messages

GPT-4o Response:

Thank you for providing the enhanced versions of the `file_processor.py` script. Both examples demonstrate thoughtful improvements and robust handling for processing text files and counting word frequencies. Here's a consolidated summary of the key features and improvements made in the scripts:

Key Features:

1. **Efficient Text Cleaning** (`clean_line` function):**
 - Uses `str.maketrans` to create a translation table for removing punctuation.
 - Converts all characters to lowercase for case-insensitive word counting.
 - Trims leading/trailing whitespace and normalizes multiple spaces to a single space.
2. **Error Handling**:**
 - Handles common errors such as `FileNotFoundError` and `UnicodeDecodeError`.
 - Provides meaningful error messages and gracefully exits the script on failure.
3. **Command-Line Argument Support**:**
 - Allows users to specify the input file via command-line arguments.
 - Defaults to `input.txt` if no argument is provided.
4. **Word Counting**:**
 - Counts words using a dictionary (`word_counts`) where each key is a word and the value is its frequency.
 - Skips empty lines and empty strings after cleaning.
5. **Advanced Features with `Counter`**:**
 - Demonstrates the use of `collections.Counter` to efficiently find the most common words.
 - Sorts words alphabetically when printing all word frequencies.
6. **Unicode and Encoding Support**:**
 - Attempts to read files as UTF-8 initially, with a fallback to `latin-1` for non-UTF-8 files.
7. **Informative Outputs**:**
 - Prints useful statistics such as the number of lines processed, the number of unique words, and the top 10 most common words.
 - Outputs all words and their counts alphabetically.
8. **Documentation and Readability**:**
 - Includes docstrings for every function to clearly explain its purpose, arguments, and return values.
 - Provides detailed comments to explain the logic and steps in the code.

Suggestions for Further Improvement:

1. **Handling Stop Words**:**

- You could add functionality to exclude common stop words (e.g., "the", "and", "is") from the word count. This can be achieved by maintaining a list or set of stop words to filter out.

```
```python
STOP_WORDS = {'the', 'and', 'is', 'in', 'to', 'of', 'a', 'that', 'it'}
...
if word not in STOP_WORDS:
 word_counts[word] = word_counts.get(word, 0) + 1
...
```
```

2. ****Output to a File**:**

- Add an option to save the word count results to an output file (e.g., `output.txt`) for easier analysis and sharing.

```
```python
with open('output.txt', 'w') as out_file:
 for word, count in sorted(counts.items()):
 out_file.write(f"{word}: {count}\n")
...
```
```

3. ****Support for Additional Punctuation**:**

- Consider expanding `clean_line` to handle additional punctuation such as dashes (`-`, `—`) and special characters like emojis or non-standard symbols.

```
```python
import re
def clean_line(line):
 line = re.sub(r'[^\w\s]', "", line) # Remove non-word characters
 return line.lower().strip()
...
```
```

4. ****Unit Tests**:**

- Add a test suite to verify the correctness of individual functions. For example, test cases for `clean_line` and `count_words` can ensure the script behaves as expected.

5. ****Performance Optimization**:**

- For very large files, consider using generators to process lines incrementally rather than loading the entire file into memory.

```
```python
def load_file(filepath):
 with open(filepath, 'r', encoding='utf-8') as f:
 for line in f:
 yield line
...
```
```

6. ****Cross-Platform Compatibility**:**

- Ensure compatibility with different operating systems by normalizing file paths using `os.path`.

7. ****Interactive Mode**:**

- Add an interactive mode where users can query word frequencies directly, e.g., "How many times does the word 'example' appear?"

Final Thoughts:

Your script demonstrates great attention to detail and a solid understanding of Python programming principles. The additional improvements suggested above can make the script even more versatile and user-friendly. Keep up the good work!

Response time: 11.39 seconds

Llama-3.1 Response:

Here are two versions of the completed script:

****Version 1****

```
```python
file_processor.py - Script for processing text files and counting word frequencies
import string
import sys
```

```
def load_file(filepath):
```

```
 """Load a text file and return its content as a list of lines.
```

```
 Args:
```

```
 filepath: Path to the text file
```

```
 Returns:
```

```
 List of strings, each representing a line in the file
```

```
 Raises:
```

```
 FileNotFoundError: If the file doesn't exist
```

```
 """
```

```
 try:
```

```
 with open(filepath, 'r', encoding='utf-8') as f:
```

```
 return f.readlines()
```

```
 except FileNotFoundError:
```

```
 print(f"Error: File '{filepath}' not found.")
```

```
 sys.exit(1)
```

```
 except Exception as e:
```

```
 print(f"Error reading file: {e}")
```

```
 sys.exit(1)
```

```
def clean_line(line):
```

```
 """Clean a line by removing punctuation and converting to lowercase.
```

```
 Args:
```

```
 line: A string representing a line of text
```

```
 Returns:
```

```

 Cleaned string with punctuation removed and all lowercase
 """

 # Remove punctuation and make lowercase
 translator = str.maketrans("", "", string.punctuation)
 return line.translate(translator).lower().strip()

def count_words(lines):
 """Count frequency of each word in the provided lines.

 Args:
 lines: List of strings, each representing a line of text

 Returns:
 Dictionary mapping words to their frequency counts
 """
 word_counts = {}
 for line in lines:
 clean = clean_line(line)
 for word in clean.split():
 if word: # Skip empty strings
 word_counts[word] = word_counts.get(word, 0) + 1
 return word_counts

def main():
 """Main function to process text file and display word counts."""
 # Get filepath from command line arguments or use default
 filepath = sys.argv[1] if len(sys.argv) > 1 else 'input.txt'

 print(f"Processing file: {filepath}")
 lines = load_file(filepath)

 counts = count_words(lines)

 if not counts:
 print("No words found in the file.")
 return

 print(f"Found {len(counts)} unique words:")
 for word, count in sorted(counts.items()):
 print(f"{word}: {count}")

if __name__ == '__main__':
 main()
...
Version 2

```python
#!/usr/bin/env python3
"""File processor script that cleans text and counts word frequencies."""

```

```

import string
import sys
import os
from collections import Counter

def load_file(filepath):
    """Load a text file and return its content as a list of lines.

    Args:
        filepath: Path to the text file to process

    Returns:
        List of strings where each string is a line from the file

    Raises:
        FileNotFoundError: If the specified file doesn't exist
    """
    if not os.path.exists(filepath):
        raise FileNotFoundError(f"File not found: {filepath}")

    try:
        with open(filepath, 'r', encoding='utf-8') as f:
            return f.readlines()
    except UnicodeDecodeError:
        # Try again with a different encoding if UTF-8 fails
        with open(filepath, 'r', encoding='latin-1') as f:
            return f.readlines()

def clean_line(line):
    """Remove punctuation from a line and convert to lowercase.

    Args:
        line: String representing a line of text

    Returns:
        Cleaned string with no punctuation and all lowercase characters
    """
    # Create a translation table to remove all punctuation
    translator = str.maketrans("", "", string.punctuation)

    # Remove punctuation, convert to lowercase, and strip whitespace
    cleaned = line.translate(translator).lower().strip()

    # Replace multiple spaces with a single space
    cleaned = ' '.join(cleaned.split())

    return cleaned

```

```

def count_words(lines):
    """Count the frequency of each word in the given lines.

    Args:
        lines: List of text lines to process

    Returns:
        Dictionary mapping words to their frequency counts
    """
    word_counts = {}
    for line in lines:
        if not line.strip(): # Skip empty lines
            continue

        clean = clean_line(line)
        for word in clean.split():
            if word: # Skip empty strings
                word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts

def parse_args():
    """Parse command line arguments.

    Returns:
        String representing the filepath to process
    """
    if len(sys.argv) > 1:
        return sys.argv[1]
    return 'input.txt' # Default filepath

def main():
    """Main function to process a text file and print word frequencies."""
    try:
        filepath = parse_args()

```

Response time: 37.11 seconds

Analysis: Both GPT-4o and Llama successfully completed the `clean_line` function and enhanced the script, but their approaches and robustness varied. In Zero-Shot, GPT-4o implemented `clean_line` using `str.maketrans` to remove punctuation, converted text to lowercase, and trimmed whitespace, maintaining simplicity but lacking error handling or input flexibility. Llama's response was similar. In Few-Shot, GPT-4o provided a detailed summary of the provided examples, highlighting features like UTF-8 encoding and Counter usage, but did not produce a completed script. Llama's few-shot response delivered two versions: a simple one mirroring Example 1 with command-line support and error handling, and a robust one incorporating Example 2's features like encoding fallbacks, Counter for top words, and empty line skipping.

Comparison: Llama performed better overall, particularly in few-shot, for its thorough integration of robust features and complete code delivery.

Summary Comparison

Model Performance:

GPT-4o consistently produced concise, reliable, and readable solutions, often leaning on libraries and idiomatic constructs. Llama 3.1 tended to be more verbose and exploratory, sometimes favoring lower-level implementations. While both models performed well, GPT-4o more frequently delivered production-ready code, especially when paired with well-structured prompts.

Prompting Techniques:

Few-shot prompting improved output quality across both models. It encouraged clearer structure, better error handling, and adherence to the desired output format. Zero-shot prompts were effective for simple tasks but often lacked robustness. Example-driven prompts had the greatest impact in tasks involving formatting, validation, and code generation.

Conclusion:

GPT-4o is better suited for high-quality, efficient code generation with minimal prompting. Llama 3.1 shows promise and excels in providing in-depth explanations, making it useful for learning and exploratory use (in some cases it delivers a better response than GPT-4o). Prompting style plays a critical role in guiding both models toward higher accuracy and completeness, particularly in real-world development contexts.