GenAI for Software Development: Assignment 3

Ted Tran tmtran03@wm.edu
Justin Liu jliu48@wm.edu

1 Introduction

Prompt Engineering for In-Context Learning explores how different prompt designs influence Large Language Models (LLMs) in solving diverse software engineering tasks. In this assignment, we applied five strategies—zero-shot, few-shot, chain-of-thought, prompt-chaining, and BLEU-based self-consistency—to 22 problems ranging from code summarization and bug fixing to API generation and code translation. Our experiments compare two models—ChatGPT (gpt-4o-mini) and Codestral-2501, demonstrating how strategic prompt examples and structured reasoning affect code quality and clarity. Temperature is set to 0.7, and the token limit is 1024 for all tables. All prompts, raw outputs, and automation scripts are available in our GitHub repository: https://github.com/tmtran03/Prompt-Engineering-LLMs.

2 Takeaways

Across tasks, few-shot prompting consistently yielded concise, high-precision solutions—ideal for practical coding—while chain-of-thought prompts delivered deeper conceptual understanding at the cost of verbosity. Prompt-chaining struck a balance, offering structured, step-by-step guidance, and BLEU-based self-consistency effectively selected the most representative output without manual review. GPT-4o-mini exhibited richer natural language and contextual nuance, especially in chain-of-thought scenarios, whereas Codestral-2501 produced leaner, example-driven responses that are quick to scan. Ultimately, prompt style drives not only correctness but also the instructional depth and usability of LLM-generated code.

3 Results

The results of our experiments can be seen below, showcasing the performance of the different prompt strategies and models across the 22 software engineering tasks. The evaluation metrics and comparative analysis highlight the strengths of each strategy, providing insights into how prompt structure influences output quality and clarity.

Task 1 - Code Summarization (Java)

Strategy:	zero_shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt	Summarize the functionality of the following Java method:
	``cpp
	public Map<String, Integer> countWordFrequency(List<String> words) {
	Map<String, Integer> freqMap = new HashMap<>();
	for (String word : words) {
	freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);
	}
	return freqMap;
	}	

Output:

The `countWordFrequency` method takes a `List<String>` of words as input and returns a `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 the word counts. 2. It iterates through each word in the provided list. 3. For each word, it updates the `freqMap` by either retrieving the current count of the word (if it exists) or defaulting to 0 if the word isn't already in the map. It then increments this count by 1. 4. Finally, it returns the `freqMap`, which contains each unique word as a key and its corresponding count as the value. In summary, this method effectively counts how many times each word appears in the input list.	The provided Java method `countWordFrequency` takes a list of strings (`List<String> words`) as input and returns a map (`Map<String, Integer>`) that counts the frequency of each word in the list. Here is a step-by-step summary of its functionality: 1. **Initialization**: It initializes an empty `HashMap` called `freqMap` to store the word frequencies. 2. **Iteration**: It iterates over each word in the input list `words`. 3. **Frequency Calculation**: - For each word, it checks if the word is already a key in the `freqMap`. - If the word is not present, it adds the word to the map with an initial count of `1`. - If the word is already present, it increments its count by `1`. 4. **Return**: Finally, it returns the `freqMap` containing the frequency of each word. In summary, the method counts how many times each word appears in the provided list and returns this information in a map where the keys are the words and the values are their respective counts.
Comment:	Clear explanation with a step-by-step breakdown. Slightly verbose but accurate and complete.

Strategy:	few_shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt	Example:
	Input: a Java method that counts the frequency of each character in a string.
	Output: This method takes a string as input and returns a map where each key is a character from the string, and the value is the number of times that character appears.
	
	Now, summarize the functionality of the following Java method:
	``cpp
	public Map<String, Integer> countWordFrequency(List<String> words) {
	Map<String, Integer> freqMap = new HashMap<>();
	for (String word : words) {
	freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);
	}
	return freqMap;
	}
	
	``	

Output:

This method takes a list of strings (words) as input and returns a map where each key is a unique word from the list,	The provided Java method, `countWordFrequency`, takes a list of strings as input and returns a map where each key is a string from the list, and

and the value is the number of times that word appears in the list.	the corresponding value is the number of times that string appears in the list. Here is a summary of its functionality: 1. **Input**: A `List<String>` containing words. 2. **Output**: A `Map<String, Integer>` where the keys are the words from the input list, and the values are the frequencies of those words. The method works as follows: - It initializes an empty `HashMap` called `freqMap` to store the word frequencies. - It iterates through each string in the input list. - For each string, it updates the `freqMap` by incrementing the count for that string using `getOrDefault`, which ensures that if the string is not already in the map, it initializes the count to 0 before incrementing. - Finally, it returns the `freqMap` containing the word frequencies. This method is useful for counting the occurrence of each word in a collection of words.
Comment:	Concise and structured, maintaining clarity while matching the style of the given example. The response feels more natural and aligned with instructional summaries.

Model Analysis and Comparison: GPT's zero-shot output is detailed and methodical, providing a comprehensive explanation suited for those unfamiliar with Java map operations. In contrast, Codestral's few-shot strategy produces a cleaner, more example-driven response that is easier to scan and more efficient for experienced readers. While both outputs are correct, zero-shot is better for foundational understanding, whereas few-shot excels in instructional alignment and brevity.

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

Strategy:	few_shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt	Example:
	Input: start=1, end=4 → Output: 6 (This sums 1+2+3)
	Now identify and fix the off-by-one error in the following function:
	``python
	def sum_range(start, end):
	total = 0
	for i in range(start, end):
	total += i
	return total
	
	``	

Output:

The function `sum_range` currently sums the numbers starting from `start` up to, but not including, `end`. This is why it produces an off-by-one error; it does not include the `end` value in the sum.	The function `sum_range` currently has an off-by-one error because it does not include the `end` value in the summation. The loop `for i in range(start, end):` iterates from `start` to `end-1`, excluding `end`. To fix this issue, you should change the loop to include the `end` value.

To fix this issue, you can modify the `range` function to include `end` by changing the `end` parameter in the `range` function to `end + 1`. Here's the corrected version of the function:
 	
``python
def sum_range(start, end):
total = 0
for i in range(start, end + 1): # Include `end` by changing `end` to `end + 1`
total += i
return total
``
 	
Now, if you call `sum_range(1, 4)`, it will correctly compute the sum \(1 + 2 + 3 + 4 = 10\).
	You can do this by changing the loop to `for i in range(start, end + 1):`.
	
	Here is the corrected function:
	
	``python
	def sum_range(start, end):
	total = 0
	for i in range(start, end + 1):
	total += i
	return total
	``
	
	With this correction, the function will properly include the `end` value in the summation. For example:
	
	``python
	print(sum_range(1, 4)) # Output: 10 (This sums 1+2+3+4)
	``
	
	This should give the correct output for the given inputs.

Comment:	The few-shot model quickly identified the off-by-one error and fixed it by including end + 1 in the range. The explanation was clear, concise, and aligned with the example provided.

Strategy:	chain_of_thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Think step-by-step to find and correct the off-by-one error in the following Python function:
	``python
	def sum_range(start, end):
	total = 0
	for i in range(start, end):
	total += i
	return total
	
	``"	

Output:

Let's analyze the given function `sum_range(start, end)` step-by-step to identify and correct any off-by-one error.
 	
### Understanding the Function 	
 	
1. **Purpose**: The function is intended to calculate the sum of all integers from `start` to `end`, inclusive.
2. **Current Implementation**: It uses a `for` loop with the `range` function, which generates numbers starting from `start` up to (but not including) `end`.
 	
### Identifying the Off-by-One Error 	
	To find and correct the off-by-one error in the given Python function `sum_range`, let's break down the logic step by step.
	
	The function `sum_range` is supposed to sum all integers from `start` to `end` (inclusive). Let's analyze the current implementation:
	
	``python
	def sum_range(start, end):
	total = 0
	for i in range(start, end):
	total += i
	return total

The range function in Python is exclusive of the end value. This means that if you call `sum_range(1, 5)`, the loop will iterate over the values `1, 2, 3, 4`, and it will **not** include `5`. Therefore, the total will be the sum of `1 + 2 + 3 + 4 = 10`, whereas the correct sum (including `5`) should be `1 + 2 + 3 + 4 + 5 = 15`.
 	
### Correcting the Function 	
 	
To include the `end` value in the summation, we can modify the `range` function to go one step beyond `end`. This can be accomplished by changing `range(start, end)` to `range(start, end + 1)`.
 	
### Updated Function 	
 	
Here's the corrected function:
 	
``python
def sum_range(start, end):
total = 0
for i in range(start, end + 1): # Include 'end' in the range
total += i
return total
``
 	
### Example Validation 	
 	
Let's validate the corrected function with an example:
- If we call `sum_range(1, 5)`, it will now correctly iterate over `1, 2, 3, 4, 5`, resulting in `15`.
 	
### Summary 	
 	
We identified that the original function excluded the `end` value due to the behavior of the `range` function. By adjusting the range to `end + 1`, we ensure that the function includes the `end` value in the summation, thus correcting the off-by-one error.
	``
	
	Here's what's happening in the function:
	
	1. We initialize `total` to 0.
	2. We use a `for` loop to iterate over the range from `start` to `end` (exclusive).
	3. For each `i` in the range, we add `i` to `total`.
	4. Finally, we return `total`.
	
	The problem is that the `range(start, end)` function generates numbers from `start` to `end - 1`. This means that `end` is not included in the sum. To include `end` in the sum, we need to change the range to `range(start, end + 1)`.
	
	Here's the corrected function:
	
	``python
	def sum_range(start, end):
	total = 0
	for i in range(start, end + 1):
	total += i
	return total
	``
	
	Now, the loop will iterate from `start` to `end` (inclusive), and all numbers in this range will be added to `total`.
	
	Let's verify the corrected function with a couple of test cases:
	
	``python
	print(sum_range(1, 5)) # Should return 1 + 2 + 3 + 4 + 5 = 15
	print(sum_range(2, 4)) # Should return 2 + 3 + 4 = 9
	``
	
	Both test cases should now work correctly, and the function will return the expected sum including the `end` value.

Comment:	The chain-of-thought model took a more detailed step-by-step approach to explain the off-by-one issue and validate the fix with examples. While slightly more verbose, it offered a thorough understanding of the logic behind the correction.

Model Analysis and Comparison: The few-shot strategy uses a targeted example to efficiently detect and fix the off-by-one error, producing a clean and quick correction ideal for practical coding tasks. The chain-of-thought approach, in contrast, thoroughly walks through the reasoning behind the bug and tests the fix with sample inputs, offering deeper conceptual clarity. GPT's output in both strategies was accurate, but the few-shot version prioritized brevity and structure, while the chain-of-thought version emphasized instructional depth. This shows that prompt style affects not only the length of the output but also its pedagogical value.

Task 3 - Bug Classification (C++)

Strategy:	chain_of_thought

Justin Liu and Ted Tran

Model:	gpt-4o-mini	Codestral-2501

Prompt	"Walk through your reasoning step-by-step and classify the bug in the following C++ function:
	``cpp
	int * getArray(int size) {
	  int arr[size]; // Warning: local array
	  return arr; // Bug: returning pointer to local variable
	}
	
	``"	

Output:

"To classify the bug in the provided C++ function step-by-step, let's analyze the code line by line:
 	
1. **Function Declaration**:
  ``cpp
  int * getArray(int size) {
  ``
  This line declares a function named `getArray` that takes an integer parameter `size` and returns a pointer to an integer.
 	
2. **Local Array Declaration**:
  ``cpp
  int arr[size]; // Warning: local array
  ``
  Here, a local array `arr` of size `size` is declared. In C++, arrays declared inside a function have automatic storage duration, meaning they are allocated on the stack. Once the function scope ends (i.e., when the function returns), the memory allocated for this array will be deallocated.
 	
3. **Returning the Local Array**:
  ``cpp
  return arr; // Bug: returning pointer to local variable
  ``
  This line attempts to return a pointer to the local array `arr`. However, since `arr` is a local variable, returning a pointer to it results in undefined behavior. Specifically, after the function returns, the memory for `arr` is deallocated, and the pointer returned will point to an area of memory that is no longer valid.
 	
4. **Classifying the Bug**:
  The bug can be classified as a **Dangling Pointer** issue. A dangling pointer occurs when a pointer references a memory location that has been deallocated. In this case:
  - The function returns a pointer to a local variable (`arr`), which goes out of scope once the function exits.
  - Accessing this pointer after the function has returned can lead to unpredictable behavior, crashes, or data corruption since the memory it points to is no longer valid.
 	
5. **Additional Considerations**:
	"Let's walk through the function step-by-step to identify the bug.
	
	### Step-by-Step Analysis
	
	1. **Function Definition:**
	  ``cpp
	  int * getArray(int size) {
	  ``
	  This function is designed to return a pointer to an array of integers. The parameter `size` specifies the number of elements in the array.
	
	2. **Local Array Declaration:**
	  ``cpp
	  int arr[size];
	  ``
	  Here, an array `arr` is declared with `size` elements. This array is local to the function `getArray`. This means that the memory for `arr` is allocated on the stack when the function is called.
	
	3. **Return Statement:**
	  ``cpp
	  return arr;
	  ``
	  The function attempts to return a pointer to the local array `arr`.
	
	### Identifying the Bug
	
	The primary issue in this function is related to the **scope and lifetime** of the local array `arr`.
	
	- **Local Scope:** The array `arr` is defined within the function `getArray`. As soon as the function `getArray` completes its execution, the local variables, including `arr`, are destroyed. Their memory is deallocated, and any pointers to this memory become invalid.
	
	- **Returning a Pointer to a Local Variable:** By returning a pointer to `arr`, the function is essentially returning a pointer to memory that will be invalid after the function returns. This leads to undefined behavior because any attempt to access the returned pointer will result in accessing deallocated memory.
	
	### Classification of the Bug

- The warning `// Warning: local array` is also relevant because it signifies that the size of the array is determined at runtime, which is not standard in C++ prior to C++11. In C++11 and later, using `std::vector<int>` or dynamically allocating the array with `new` would be more appropriate to avoid such issues.
- If you want to allocate an array dynamically and return a pointer safely, you could do it like this:

```cpp
int * getArray(int size) {
    return new int[size]; // Dynamically allocate memory
}
```

However, in this case, the user of the function must remember to `delete[]` the allocated memory to prevent memory leaks.

In summary, the bug in the code is a **Dangling Pointer** issue caused by returning a pointer to a local variable that goes out of scope when the function returns."

This bug is classified as:

- **Dangling Pointer:** The returned pointer points to memory that is no longer valid (deallocated) after the function exits.
- **Undefined Behavior:** Accessing the returned pointer can lead to unpredictable behavior, such as crashes, incorrect data, or security vulnerabilities.

Solution

To fix this bug, you should dynamically allocate the array using `new` or `malloc`, and ensure that the memory is properly freed when it is no longer needed. Here is an example using `new`:

```cpp
int * getArray(int size) {
    int *arr = new int[size]; // Dynamically allocate memory
    return arr; // Return the pointer to the dynamically allocated array
}
```

Make sure to properly manage the memory by deleting the array when it is no longer needed to avoid memory leaks:

```cpp
int main() {
    int size = 10;
    int *arr = getArray(size);

    // Use arr...

    delete[] arr; // Free the dynamically allocated memory
    return 0;
}
```

Alternatively, you can use `std::vector` from the C++ Standard Library, which automatically handles memory management for you:

```cpp
#include <vector>

std::vector<int> getArray(int size) {
    std::vector<int> arr(size); // Use std::vector for automatic memory management
    return arr; // Return the vector
}

int main() {
    int size = 10;
    std::vector<int> arr = getArray(size);

    // Use arr...

    return 0;
```

	}
	``
	
	Using `std::vector` is generally more safe and convenient because it handles memory allocation and deallocation automatically."

Comment:	Clear and methodical explanation that classifies the bug as a dangling pointer. It provides context on memory lifetime and explains the consequences thoroughly, including undefined behavior and alternative safe implementations.

Strategy:	prompt_chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step 1: Identify the bug in the following C++ function.
	Step 2: Explain why it occurs and its consequences.
	``cpp
	int * getArray(int size) {
	  int arr[size]; // Warning: local array
	  return arr; // Bug: returning pointer to local variable
	}
	
	``"	

Output:

"### Step 1: Identify the bug
 	
The bug in the provided C++ function is that it returns a pointer to a local variable `arr`, which is defined as a local array within the function:
 	
``cpp
int arr[size]; // Local array
return arr; // Returning pointer to local variable
``
 	
### Step 2: Explain why it occurs and its consequences 	
 	
#### Why it occurs: 	
In C++, local variables (including local arrays) are allocated on the stack. When the function `getArray` exits, the memory allocated for `arr` is freed and the stack frame is destroyed. As a result, the pointer returned by the function becomes a dangling pointer, pointing to memory that is no longer valid.
 	
Here's what happens in detail:
- The array `arr` is created on the stack when the function is called.
- The function returns a pointer to this array.
- Once the function exits, the stack frame is popped off, and `arr` is no longer accessible. The memory that was occupied
	"Sure, let's break down the problem step by step.
	
	### Step 1: Identify the Bug
	
	The bug in the provided C++ function is related to the scope of the local variable `arr`. The function `getArray` is designed to return a pointer to an array of integers. However, the array `arr` is declared locally within the function, which means it has automatic storage duration and will be destroyed when the function returns. Returning a pointer to this local array is problematic because the memory it points to is no longer valid after the function call.
	
	Here is the problematic code:
	``cpp
	int * getArray(int size) {
	  int arr[size]; // Warning: local array
	  return arr; // Bug: returning pointer to local variable
	}
	``
	
	### Step 2: Explain Why It Occurs and Its Consequences
	
	#### Why It Occurs
	
	1. **Automatic Storage Duration**: The array `arr` is declared within the function `getArray`. This means `arr` has automatic storage duration, which means it is allocated on the stack and will be destroyed when the

by `arr` can be reused for other purposes.

Consequences:
The consequences of returning a pointer to a local variable are severe:
1. **Undefined Behavior**: Any attempt to access the memory pointed to by this pointer after the function has returned leads to undefined behavior. This could manifest as reading garbage values, crashes, or corruption of other data.
2. **Memory Leaks**: If the caller of the function mistakenly believes that they own the memory and tries to free it, it could lead to double-free errors or memory leaks, depending on how the memory is managed.
3. **Logic Errors**: If the program relies on the data stored in that memory, it may lead to incorrect results or program states, which can be very difficult to debug.

Suggested Fix:
To correctly return an array, you can use dynamic memory allocation, or better yet, use standard containers like `std::vector` which handle memory management automatically:

Using dynamic memory allocation:
```cpp
int* getArray(int size) {
    return new int[size]; // Allocate memory on the heap
}
```

Or using `std::vector`:
```cpp
#include <vector>

std::vector<int> getArray(int size) {
    return std::vector<int>(size); // Returns a vector, managing memory automatically
}
```

Both of these alternatives avoid the issue of dangling pointers and ensure that the memory remains valid for the duration of its intended use."

function returns.

2. **Returning a Pointer to a Local Variable**: The function returns a pointer to the local array `arr`. When `getArray` returns, the memory for `arr` is deallocated, and the pointer becomes invalid (dangling pointer).

Consequences

1. **Undefined Behavior**: Accessing the memory through the returned pointer after the function call results in undefined behavior. This means the program might crash, produce incorrect results, or exhibit other unexpected behaviors.

2. **Security Vulnerabilities**: Returning a pointer to a local variable can expose the program to security vulnerabilities such as buffer overflows, which can be exploited by malicious code.

3. **Memory Leaks**: While not directly a memory leak, the invalid use of the returned pointer can lead to resource management issues, which can indirectly contribute to memory leaks.

Correcting the Code

To fix the issue, you should allocate the array dynamically using `new` and ensure that the caller of the function is responsible for deallocating the memory using `delete[]`. Here is the corrected version of the function:

```cpp
int * getArray(int size) {
    int *arr = new int[size]; // Dynamically allocate array
    return arr;               // Return pointer to dynamically allocated array
}
```

And the caller should handle the deallocation like this:

```cpp
int main() {
    int size = 10;
    int *arr = getArray(size);

    // Use the array...

    delete[] arr; // Deallocate the dynamically allocated array
    return 0;
}
```

By dynamically allocating the array, you ensure that the memory remains valid until it is explicitly deallocated by the caller."

Comment	The reasoning is structured and covers both identification and consequences step-by-step. It emphasizes undefined behavior and potential security issues, with multiple alternatives suggested, showing a deeper awareness of safe memory handling.

Model Analysis and Comparison: The chain-of-thought strategy encourages deep reasoning, walking through memory behavior and clearly explaining why the bug occurs. It's ideal for learning and debugging, but more verbose. Prompt chaining structures the explanation into labeled steps, making it clearer and more digestible for quick understanding. Both models correctly identify the dangling pointer, but chain-of-thought favors depth, while prompt chaining excels in clarity and usability.

Task 4 - Generating Email Validators (Python + Regex)

Strategy:	prompt_chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step 1: Provide a regex pattern for basic email addresses (e.g., user@domain.com).
	Step 2: Implement `is_valid_email` using that pattern in the following code:
	``python
	def is_valid_email(email):
	  # TODO: Complete using regex
	  pass
	
	``"	

Output:

"To create a regex pattern for basic email addresses and implement the `is_valid_email` function, we can follow these steps:
 	
### Step 1: Define a Regex Pattern 	
 	
A basic regex pattern for email addresses can be defined as follows:
- The local part (before the `@`) can include letters, digits, dots, underscores, and hyphens.
- The domain part (after the `@`) typically consists of letters, followed by a dot and a domain suffix (like `.com`, `.org`, etc.).
 	
Here is a simple regex pattern that captures these rules:
 	
``regex
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
``
 	
### Step 2: Implement `is_valid_email` 	
 	
Now we can implement the `is_valid_email` function using the regex pattern defined above. We will use Python's `re` module to match the email against our regex.
 	
Here's the complete code:
 	
``python
import re
 	
def is_valid_email(email):
  # Regex pattern for validating an email address
	"Sure, let's first create a regex pattern for a basic email address and then implement the `is_valid_email` function using that pattern.
	
	### Step 1: Provide a regex pattern for basic email addresses
	
	A typical regex pattern for basic email addresses can be:
	
	``regex
	^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
	``
	
	Here's a breakdown of the pattern:
	- `^` asserts the position at the start of the string.
	- `[a-zA-Z0-9._%+-]+` matches one or more characters that are either letters, digits, or one of the specified special characters `.`, `_`, `%`, `+`, or `-`.
	- `@` matches the literal `@` character.
	- `[a-zA-Z0-9.-]+` matches one or more characters that are either letters, digits, `.`, or `-`.
	- `\.` matches the literal `.` character.
	- `[a-zA-Z]{2,}` matches two or more letters, representing the domain suffix (e.g., `.com`, `.org`).
	- `$` asserts the position at the end of the string.
	
	### Step 2: Implement `is_valid_email` using the provided pattern
	
	Now, let's implement the `is_valid_email` function:
	
	``python
	import re
	
	def is_valid_email(email):
	  # Define the regex pattern for basic email addresses

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 re.match(pattern, email) is not None
 	
# Example usage 	
print(is_valid_email(""user@example.com"")) # True
print(is_valid_email(""invalid-email"")) # False
``
 	
### Explanation: 	
- The regex pattern checks for a combination of valid characters in the local part followed by `@`, then a valid domain name and a top-level domain (TLD).
- The `re.match` function checks if the entire string matches the pattern, returning `True` if it does and `False` otherwise."
	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
	if re.match(pattern, email):
	return True
	else:
	return False
	
	# Example usage
	print(is_valid_email(""user@domain.com"")) # True
	print(is_valid_email(""user@domain"")) # False
	print(is_valid_email(""user@domain.c"")) # False
	print(is_valid_email(""user@.com"")) # False
	``
	
	This `is_valid_email` function will return `True` if the email matches the basic email pattern and `False` otherwise."

Comment:	The prompt chaining strategy gives a well-balanced and standard regex suited for common email validation tasks. It clearly explains the components of the pattern and results in a clean, easy-to-implement function that works for general use cases.

Strategy:	chain_of_thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Think step-by-step to derive a regex and implement `is_valid_email` in the following code:
	``python
	def is_valid_email(email):
	# TODO: Complete using regex
	pass
	
	``"	

Output:

"To derive a regex for validating email addresses and implement the `is_valid_email` function, we need to follow a systematic approach. Let's break down the requirements for a valid email address and then construct a regex pattern accordingly. ### Step 1: Understand the Structure of an Email Address An email address generally consists of two main parts: 1. **Local Part**: This is the part before the ""@"" symbol. It can contain: - Letters (both uppercase and lowercase) - Digits - Special characters (like `.`, `-`, `_`, `+`) - It should not start or end with a special character, nor should there be consecutive special characters.	"To implement the `is_valid_email` function using regex, we need to define a pattern that matches the structure of a valid email address. Here's a step-by-step approach to derive the regex pattern and then implement the `is_valid_email` function: ### Step-by-Step Derivation of the Regex Pattern 1. **Local Part**: - Must start with an alphanumeric character or an underscore (`[a-zA-Z0-9_]`). - Can be followed by any combination of alphanumeric characters, dots (`.`), underscores, or hyphens (`[a-zA-Z0-9_.-]*`). - Should not end with a dot (`.`). Combined, the local part can be represented as: ``` [a-zA-Z0-9_][a-zA-Z0-9_.-]*[a-zA-Z0-9_]

2. **Domain Part**: This is the part after the ""@"" symbol. It typically includes:
   - A domain name (like `example` in `example.com`)
   - A top-level domain (like `.com`, `.org`, etc.), which must be at least two characters long.

Step 2: Create the Regex Pattern

Based on the above understanding, we can start constructing the regex pattern:

1. **Local Part Regex**:
   - Can start with a letter or digit.
   - Can include letters, digits, and special characters.
   - Cannot have consecutive special characters.
   - Cannot end or start with a special character.

   A possible regex for the local part can be:
   ```
   [a-zA-Z0-9](?:[a-zA-Z0-9._%+-]*[a-zA-Z0-9])?
   ```

2. **Domain Part Regex**:
   - Should start with a letter or digit.
   - Can contain letters, digits, and hyphens but not start or end with a hyphen.
   - The top-level domain must consist of at least two letters.

   A possible regex for the domain part can be:
   ```
   [a-zA-Z0-9]([a-zA-Z0-9-]*[a-zA-Z0-9])?\.[a-zA-Z]{2,}
   ```

3. **Combining the Two Parts**:
   We combine the local and domain parts with an ""@"" in between:
   ```regex

^[a-zA-Z0-9](?:[a-zA-Z0-9._%+-]*[a-zA-Z0-9])?@[a-zA-Z0-9]([a-zA-Z0-9-]*[a-zA-Z0-9])?\.[a-zA-Z]{2,}$
   ```

Step 3: Implement the `is_valid_email` Function

Now that we have our regex pattern, we can implement the `is_valid_email` function using Python's `re` module.

Here's how the final implementation looks:

```python
import re

def is_valid_email(email):
    # Define the regex pattern for validating an email
   ```

2. **Domain Part**:
   - Typically consists of domain labels separated by dots (`.`).
   - Each label can start and end with alphanumeric characters (`[a-zA-Z0-9]`).
   - Each label can also contain hyphens (`-`) in between but not at the start or end.
   - The top-level domain (TLD) is usually 2-6 characters long.

   Combined, the domain part can be represented as:
   ```
   [a-zA-Z0-9-]+\.[a-zA-Z]{2,6}
   ```

3. **Combining Local and Domain Parts**:
   - The local part and the domain part are separated by an `@` symbol.

   The complete regex pattern for a valid email address is:
   ```

^[a-zA-Z0-9_][a-zA-Z0-9_.-]*[a-zA-Z0-9_]@[a-zA-Z0-9-]+\.[a-zA-Z]{2,6}$
   ```

Implementation in Python

Now, let's implement the `is_valid_email` function using the derived regex pattern.

```python
import re

def is_valid_email(email):
    # Define the regex pattern for a valid email
    pattern = r'^[a-zA-Z0-9_][a-zA-Z0-9_.-]*[a-zA-Z0-9_]@[a-zA-Z0-9-]+\.[a-zA-Z]{2,6}$'

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

# Example usage:
print(is_valid_email(""example@example.com""))  # True
print(is_valid_email(""invalid-email@.com""))   # False
print(is_valid_email(""user@sub.domain.com""))  # True
print(is_valid_email(""user@domain""))          # False
```

Explanation of the Code
1. **Import the `re` module**: This module provides support for regular expressions in Python.
2. **Define the regex pattern**: The pattern is a string that matches the

pattern = r'^[a-zA-Z0-9](?:[a-zA-Z0-9._%+-]*[a-zA-Z0-9])?@[a-zA-Z0-9]([a-zA-Z0-9-]*[a-zA-Z0-9])?\.[a-zA-Z]{2,}$' # Use re.match to check if the email matches the regex pattern return re.match(pattern, email) is not None # Examples of usage print(is_valid_email(""test@example.com"")) # True print(is_valid_email(""invalid-email@"")) # False print(is_valid_email(""user@domain.co"")) # True print(is_valid_email(""user@domain.c"")) # False print(is_valid_email(""user@domain..com"")) # False ``` ### Summary We defined a regex pattern that captures the structure of a valid email address, then implemented the `is_valid_email` function to utilize this pattern, returning `True` for valid emails and `False` for invalid ones. The provided examples demonstrate how to use the function effectively."	structure of a valid email address. 3. **Use `re.match`**: This function checks if the entire string matches the pattern. 4. **Return `True` or `False`**: Based on whether the email matches the pattern. This implementation ensures that the email is validated according to the standard email format rules."
Comment:	The chain-of-thought model goes into deeper detail, constructing the regex in stages while explaining edge cases like preventing invalid characters or structure. The final regex is more precise and restrictive, catching more subtle invalid formats.

Model Analysis and Comparison: Both strategies correctly generated functional email validators. Prompt chaining offers a quicker, standard solution that is easy to use and understand, making it ideal for general applications. The chain-of-thought strategy, while more verbose, produces a more robust regex with greater attention to detail. For more stringent validation, the chain-of-thought approach is superior, though the prompt chaining version is sufficient for most typical needs.

Task 5 - Generating Flask APIs (Python)

Strategy:	zero_shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt	“Create a Flask `/greet/<username>` endpoint that returns a JSON greeting.
	``from flask import Flask, jsonify
	app = Flask(__name__)
	
	@app.route('/greet/<username>')
	def greet(username):
	return jsonify({'greeting': f'Hello, {username}!'})```”	

Output:

"To create a Flask endpoint that returns a JSON greeting for a given username, you'll need to follow these steps:	"Certainly! Below is an example of how you can create a Flask application with an endpoint `/greet/<username>` that returns a JSON greeting

1. Install Flask if you haven't already. You can do this using pip:

   ```bash
   pip install Flask
   ```

2. Create a new Python script (e.g., `app.py`) and add the following code:

```python
from flask import Flask, jsonify

app = Flask(__name__)

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

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

Explanation of the Code:

- **Importing Modules**: The code imports the `Flask` class and `jsonify` function from the `flask` module.
- **Creating a Flask App**: An instance of the `Flask` class is created.
- **Defining the Route**: The `@app.route` decorator defines the `/greet/<username>` endpoint. The `<username>` part is a variable path parameter that Flask captures and passes to the `greet` function.
- **Creating the Response**: Inside the `greet` function, a greeting message is created using the `username` parameter. This message is then returned as a JSON response using `jsonify`.
- **Running the Application**: The application runs in debug mode when executed as the main program.

Running the Application

To run the application, execute the following command in your terminal:

```bash
python app.py
```

Testing the Endpoint

Once the server is running, you can test the `/greet/<username>` endpoint by visiting the following URL in your web browser or using a tool like `curl` or Postman:

```
http://127.0.0.1:5000/greet/YourUsername
```

For example, if you replace `YourUsername` with `Alice`, you would visit:

message.

First, ensure you have Flask installed. If not, you can install it using pip:

```bash
pip install Flask
```

Now, create a new Python file (e.g., `app.py`) and add the following code:

```python
from flask import Flask, jsonify

app = Flask(__name__)

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

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

Explanation:
1. **Import Flask and jsonify**: These are necessary to create the Flask application and return JSON responses, respectively.
2. **Create a Flask app instance**: `app = Flask(__name__)` initializes the Flask application.
3. **Define the `/greet/<username>` endpoint**: The `@app.route` decorator specifies the URL pattern and HTTP method. The `greet` function takes `username` as an argument, constructs a greeting message, and returns it as a JSON response using `jsonify`.
4. **Run the app**: The `if __name__ == '__main__':` block ensures that the app runs only when the script is executed directly.

To run the application, execute the following command in your terminal:

```bash
python app.py
```

The Flask development server will start, and you can access the `/greet/<username>` endpoint by navigating to `http://127.0.0.1:5000/greet/<username>` in your

``
http://127.0.0.1:5000/greet/Alice
``
 	
The response will be a JSON object like this:
 	
``json
{
  ""message"": ""Hello, Alice!""
}
``
 	
That's it! You now have a simple Flask application with a `/greet/<username>` endpoint that returns a JSON greeting."
	web browser or using a tool like `curl` or Postman.
	
	For example, navigating to `http://127.0.0.1:5000/greet/John` will return:
	
	``json
	{
	  ""message"": ""Hello, John!""
	}
	``"

Comment:	The zero-shot response correctly implements the /greet/<username> endpoint with Flask and provides clear documentation. The explanation is verbose but thorough, making it beginner-friendly and easy to follow.

Strategy:	self_consistency	
Model:	gpt-4o-mini	Codestral-2501

Prompt	“Create a Flask `/greet/<username>` endpoint that returns a JSON greeting.
	``from flask import Flask, jsonify
	app = Flask(__name__)
	
	@app.route('/greet/<username>')
	def greet(username):
	  return jsonify({'greeting': f'Hello, {username}!'})```”	

Output:

"To create a Flask endpoint that returns a JSON greeting based on a username provided in the URL, you'll need to set up a Flask application and define the `/greet/<username>` route. Below is a step-by-step guide on how to achieve this.
 	
### Step 1: Install Flask 	
 	
If you haven't already installed Flask, you can do so using pip:
 	
``bash
pip install Flask
``
 	
### Step 2: Create the Flask Application 	
 	
Create a new Python file, for example, `app.py`, and add the following code:
 	
``python
from flask import Flask, jsonify
 	
app = Flask(__name__)
	"Certainly! Below is an example of how you can create a Flask application with an endpoint `/greet/<username>` which returns a JSON greeting.
	
	First, you need to ensure you have Flask installed. You can install it using pip if you haven't already:
	
	``sh
	pip install Flask
	``
	
	Next, create a Python file (e.g., `app.py`) and add the following code:
	
	``python
	from flask import Flask, jsonify
	
	app = Flask(__name__)

@app.route('/greet/<username>', methods=['GET'])
def greet(username):
  greeting = {
  'message': f'Hello, {username}!'
  }
  return jsonify(greeting)
 	
if __name__ == '__main__':
  app.run(debug=True)
``
 	
### Explanation of the Code 	
 	
1. **Importing Flask and jsonify**: We import `Flask` to create the app and `jsonify` to return JSON responses.
2. **Creating a Flask App**: We create an instance of the `Flask` class.
3. **Defining the Route**: The `@app.route('/greet/<username>', methods=['GET'])` decorator creates a route that responds to `GET` requests at the specified URL. The `<username>` portion of the URL is a variable that will be passed to the `greet` function.
4. **Creating the Greeting**: Inside the `greet` function, we create a greeting message and return it in JSON format using `jsonify`.
5. **Running the App**: The `if __name__ == '__main__':` block runs the Flask app in debug mode.
 	
### Step 3: Run the Application 	
 	
Run your Flask application from the terminal:
 	
``bash
python app.py
``
 	
### Step 4: Test the Endpoint 	
 	
You can test the endpoint using a web browser or a tool like `curl` or Postman. For example, if you run the app locally, you can access:
 	
``
http://127.0.0.1:5000/greet/John
``
 	
This should return a JSON response like:
 	
``json
{
  ""message"": ""Hello, John!""
}
``
 	
### Conclusion 	
 	
You have now created a simple Flask application with a `/greet/<username>` endpoint that returns a JSON greeting based on the username provided in the URL. You can extend this application further by adding more functionality or endpoints as needed."
	@app.route('/greet/<username>', methods=['GET'])
	def greet(username):
	  response = {
	  ""message"": f""Hello, {username}!""
	  }
	  return jsonify(response)
	
	if __name__ == '__main__':
	  app.run(debug=True)
	``
	
	Explanation:
	- We import the necessary modules from Flask.
	- We create an instance of the Flask class.
	- We define a route `/greet/<username>` which is a dynamic route that accepts a `username` parameter.
	- The `greet` function constructs a greeting message and returns it as a JSON response using `jsonify`.
	- Finally, we run the Flask app in debug mode.
	
	To run the application, execute the following command in your terminal:
	
	``sh
	python app.py
	``
	
	You should see output indicating that the Flask development server is running. You can then open a web browser or use a tool like `curl` or Postman to test the endpoint:
	
	``sh
	curl http://127.0.0.1:5000/greet/John
	``
	
	You should get a JSON response like:
	
	``json
	{
	  ""message"": ""Hello, John!""
	}
	``
	
	This simple example demonstrates how to create a dynamic endpoint in Flask that returns a JSON response."

Comment:	The self-consistency approach delivers nearly identical functionality but includes minor stylistic differences and occasionally reorders explanation for clarity. The code is consistently accurate, with slight differences in variable naming or JSON structure formatting.

Model Analysis and Comparison: Both strategies successfully implemented a working Flask endpoint returning a JSON greeting. The zero-shot approach emphasized step-by-step setup and offered more handholding, which is ideal for newcomers. The self-consistency outputs were concise, consistently accurate, and slightly more polished. Overall, both are effective, but self-consistency yields more uniform and refined results across repeated generations.

Task 6 - SQL Schema Design (SQL)

Strategy:	zero_shot	
Model:	gpt-4o-mini	Codestral-2501
Prompt:	Write SQL schema (tables, primary & foreign keys) 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)	

Output:

"Here's an SQL schema for a review app that includes tables for users, books, and reviews. Each table includes primary keys and appropriate foreign keys to establish relationships between them.

```sql
-- Users table to store information about users
CREATE TABLE Users (
    user_id SERIAL PRIMARY KEY,  -- Unique identifier for each user
    username VARCHAR(50) NOT NULL UNIQUE,  -- Unique username for each user
    email VARCHAR(100) NOT NULL UNIQUE,  -- Unique email for each user
    password_hash VARCHAR(255) NOT NULL,  -- Hashed password for security
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
-- Timestamp of account creation
);

-- Books table to store information about books
CREATE TABLE Books (
    book_id SERIAL PRIMARY KEY,  -- Unique identifier for each book
    title VARCHAR(255) NOT NULL,  -- Title of the book
    author VARCHAR(100) NOT NULL,  -- Author of the book
    published_date DATE,  -- Date the book was published
    isbn VARCHAR(20) UNIQUE,  -- ISBN number for the book
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
-- Timestamp of book entry creation
);

-- Reviews table to store user reviews of books
CREATE TABLE Reviews (
```

"To design a schema for a review app with users, books, and reviews, you can follow the structure below. This schema includes tables for users, books, and reviews, along with appropriate primary and foreign keys to establish relationships between them.

Tables and Schema

1. **Users Table**
   - Stores information about users.

2. **Books Table**
   - Stores information about books.

3. **Reviews Table**
   - Stores information about reviews, linking users and books.

SQL Schema

```sql
-- Users Table
CREATE TABLE Users (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Books Table
CREATE TABLE Books (
    book_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    author VARCHAR(100) NOT NULL,
```

```
    review_id SERIAL PRIMARY KEY,  -- Unique identifier for each review
    user_id INT NOT NULL,  -- ID of the user who wrote the review
    book_id INT NOT NULL,  -- ID of the book being reviewed
    rating INT CHECK (rating >= 1 AND rating <= 5),  -- Rating from 1 to 5
    review_text TEXT,  -- Text of the review
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  -- Timestamp of review creation
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCADE,  -- Foreign key constraint referencing Users
    FOREIGN KEY (book_id) REFERENCES Books(book_id) ON DELETE CASCADE  -- Foreign key constraint referencing Books
);

-- Indexes for performance optimization (optional)
CREATE INDEX idx_user_id ON Reviews(user_id);
CREATE INDEX idx_book_id ON Reviews(book_id);
```

Explanation of the Schema

1. **Users Table**:
   - `user_id`: A unique identifier for each user (Primary Key).
   - `username`: A unique username for each user to log in.
   - `email`: A unique email address for each user.
   - `password_hash`: A hashed version of the password for security.
   - `created_at`: A timestamp for when the user account was created.

2. **Books Table**:
   - `book_id`: A unique identifier for each book (Primary Key).
   - `title`: The title of the book.
   - `author`: The author of the book.
   - `published_date`: The date the book was published.
   - `isbn`: The ISBN number of the book (should be unique).
   - `created_at`: A timestamp for when the book entry was created.

3. **Reviews Table**:
   - `review_id`: A unique identifier for each review (Primary Key).
   - `user_id`: A foreign key referencing the `user_id` in the Users table.
   - `book_id`: A foreign key referencing the `book_id` in the Books table.
   - `rating`: An integer representing the rating given to the book (must be between 1 and 5).
   - `review_text`: A text field for the review content.
   - `created_at`: A timestamp for when the review was created.

Foreign Key Constraints
- The `user_id` in the Reviews table references the Users table, ensuring that each review is associated with a valid user.
- The `book_id` in the Reviews table references the Books table, ensuring that each review is associated with a valid book.
- The `ON DELETE CASCADE` option means that if a user or

```
    published_year INT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Reviews Table
CREATE TABLE Reviews (
    review_id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT NOT NULL,
    book_id INT NOT NULL,
    rating INT CHECK (rating >= 1 AND rating <= 5),
    comment TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCADE,
    FOREIGN KEY (book_id) REFERENCES Books(book_id) ON DELETE CASCADE
);
```

Explanation

1. **Users Table**:
   - `user_id`: Unique identifier for each user (Primary Key).
   - `username`: User's unique username.
   - `email`: User's unique email.
   - `password_hash`: Secure hash of the user's password.
   - `created_at`: Timestamp of when the user was created.

2. **Books Table**:
   - `book_id`: Unique identifier for each book (Primary Key).
   - `title`: Title of the book.
   - `author`: Author of the book.
   - `published_year`: Year the book was published.
   - `created_at`: Timestamp of when the book record was created.

3. **Reviews Table**:
   - `review_id`: Unique identifier for each review (Primary Key).
   - `user_id`: Foreign key referencing the `Users` table.
   - `book_id`: Foreign key referencing the `Books` table.
   - `rating`: Rating given to the book (1-5).
   - `comment`: Optional text comment for the review.
   - `created_at`: Timestamp of when the review was created.

Relationships

- **Users and Reviews**: A user can write multiple reviews, but each review is written by a single user. This is represented by the foreign key `user_id` in the `Reviews` table.
- **Books and Reviews**: A book can have multiple reviews, but each review is for a single book. This is represented by the foreign key `book_id` in the `Reviews` table.

Additional Considerations

- **Cascade Delete**: The `ON DELETE CASCADE` clause ensures

book is deleted, all associated reviews will also be deleted automatically. ### Indexes - Indexes on `user_id` and `book_id` in the Reviews table can enhance query performance when searching for reviews by specific users or books."	that if a user or a book is deleted, all related reviews are also deleted automatically. - **Constraints**: The `CHECK` constraint ensures that the rating is between 1 and 5. This schema provides a basic structure for a review app and can be further enhanced based on specific requirements, such as adding more user details, book details, or additional review metadata."
Comment:	The zero-shot approach delivers a focused schema tailored exactly to the prompt: users, books, and reviews. It properly defines primary and foreign keys, uses CHECK constraints, and includes performance-conscious indexing. Clean, efficient, and scoped perfectly for a book review app.

Strategy:	self_consistency	
Model:	gpt-4o-mini	Codestral-2501
Prompt	Write SQL schema (tables, primary & foreign keys) 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)	

Output:

"Creating a SQL schema for a review app involves defining the necessary tables to manage users, products (or services), reviews, and possibly categories or tags for organization. Below is an example of a SQL schema that captures these aspects.
 	
### SQL Schema for Review App 	
 	
``sql
-- Users Table
CREATE TABLE users (
  user_id SERIAL PRIMARY KEY,
  username VARCHAR(50) NOT NULL UNIQUE,
  email VARCHAR(100) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
 	
-- Products Table
CREATE TABLE products (
  product_id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  description TEXT,
  price DECIMAL(10, 2) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
 	
-- Reviews Table
	"Certainly! Below is an example of an SQL schema for a review application. This schema includes tables for users, products, reviews, and categories.
	
	``sql
	-- Users Table
	CREATE TABLE Users (
	  user_id INT AUTO_INCREMENT PRIMARY KEY,
	  username VARCHAR(50) NOT NULL UNIQUE,
	  email VARCHAR(100) NOT NULL UNIQUE,
	  password_hash VARCHAR(255) NOT NULL,
	  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
	);
	
	-- Categories Table
	CREATE TABLE Categories (
	  category_id INT AUTO_INCREMENT PRIMARY KEY,
	  name VARCHAR(100) NOT NULL UNIQUE,
	  description TEXT
	);
	
	-- Products Table
	CREATE TABLE Products (
	  product_id INT AUTO_INCREMENT PRIMARY KEY,
	  name VARCHAR(255) NOT NULL,

```
CREATE TABLE reviews (
    review_id SERIAL PRIMARY KEY,
    product_id INT NOT NULL,
    user_id INT NOT NULL,
    rating INT CHECK (rating >= 1 AND rating <= 5),
    review_text TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES products(product_id) ON DELETE
CASCADE,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);

-- Categories Table
CREATE TABLE categories (
    category_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE
);

-- Product Categories Table (Many-to-Many Relationship)
CREATE TABLE product_categories (
    product_id INT NOT NULL,
    category_id INT NOT NULL,
    PRIMARY KEY (product_id, category_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id) ON DELETE
CASCADE,
    FOREIGN KEY (category_id) REFERENCES categories(category_id) ON
DELETE CASCADE
);

-- Tags Table
CREATE TABLE tags (
    tag_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE
);

-- Review Tags Table (Many-to-Many Relationship)
CREATE TABLE review_tags (
    review_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (review_id, tag_id),
    FOREIGN KEY (review_id) REFERENCES reviews(review_id) ON DELETE
CASCADE,
    FOREIGN KEY (tag_id) REFERENCES tags(tag_id) ON DELETE CASCADE
);
```

Explanation of Tables

1. **Users Table**: Stores user information such as username, email, and password hash for authentication.

2. **Products Table**: Contains details about products (or services) including name, description, and price.

```
    description TEXT,
    price DECIMAL(10, 2) NOT NULL,
    category_id INT,
    FOREIGN KEY (category_id) REFERENCES
Categories(category_id),
    created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
);

-- Reviews Table
CREATE TABLE Reviews (
    review_id INT AUTO_INCREMENT
PRIMARY KEY,
    user_id INT,
    product_id INT,
    rating INT CHECK (rating BETWEEN 1 AND
5),
    comment TEXT,
    created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES
Users(user_id),
    FOREIGN KEY (product_id) REFERENCES
Products(product_id)
);
```

Explanation:
1. **Users Table**:
  - `user_id`: Unique identifier for each user.
  - `username`: Unique username for the user.
  - `email`: Unique email address for the user.
  - `password_hash`: Hashed password for security.
  - `created_at`: Timestamp when the user was created.

2. **Categories Table**:
  - `category_id`: Unique identifier for each category.
  - `name`: Unique name of the category.
  - `description`: Optional description of the category.

3. **Products Table**:
  - `product_id`: Unique identifier for each product.
  - `name`: Name of the product.
  - `description`: Description of the product.
  - `price`: Price of the product.
  - `category_id`: Foreign key referencing the `Categories` table.
  - `created_at`: Timestamp when the product was created.

4. **Reviews Table**:
  - `review_id`: Unique identifier for each review.

3. **Reviews Table**: Holds the reviews written by users for products, including a rating (1-5) and the review text. 4. **Categories Table**: Stores different categories that products can belong to. 5. **Product Categories Table**: A junction table for the many-to-many relationship between products and categories. 6. **Tags Table**: Contains tags that can be associated with reviews for better filtering and searching. 7. **Review Tags Table**: A junction table for the many-to-many relationship between reviews and tags. ### Additional Considerations - You may want to implement additional features such as user roles (admin, moderator), review moderation, or upvoting/downvoting reviews, which would require more tables and relationships. - Indexing key fields can improve query performance, especially for large datasets. - Make sure to handle sensitive data (like password hashes) properly and consider data privacy regulations in your application design."	- `user_id`: Foreign key referencing the `Users` table. - `product_id`: Foreign key referencing the `Products` table. - `rating`: Rating given by the user (between 1 and 5). - `comment`: Optional comment provided by the user. - `created_at`: Timestamp when the review was created. This schema provides a basic structure for a review application and can be extended further based on specific requirements."
Comment	The self-consistency strategy expands the domain from books to general products and adds advanced schema features like tags and categories with many-to-many relationships. It's more versatile and future-ready but deviates slightly from the original book-specific prompt.

Model Analysis and Comparison: The zero-shot strategy performs best for a narrowly defined requirement — it stays on topic and produces a solid, ready-to-use schema for a book review app. Self-consistency shines in generating a more flexible and extensible design suitable for broader e-commerce or review platforms. In summary, use zero-shot for precision and domain adherence, and self-consistency when extensibility or richer relationships are required.

Task 7 - Null Dereference Detection (Java)

Strategy:	few_shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Example:
	Input: a method with a null check before length().
	Output: Checks if the string is null before calling length to avoid null-dereference.
	Now identify null-dereference risk in the following code:
	``cpp
	public int getLength(String s) {
	  return s.length(); // What if s is null?
	}
	
	``"	

Output:

Checks if the string is null before calling length to avoid null-dereference.	"The code snippet provided has a potential null-dereference risk. Specifically, the method `getLength` calls the `length()` method on the

	string `s` without first checking if `s` is `null`. If `s` is `null` when `getLength` is called, attempting to call `length()` on a `null` reference will throw a `NullPointerException`.
	
	To mitigate this risk, you should add a null check before calling `length()`. Here is the revised code:
	
	``cpp
	public int getLength(String s) {
	  if (s == null) {
	  throw new IllegalArgumentException(""Input string cannot be null"");
	  }
	  return s.length();
	}
	``
	
	Alternatively, you could return a default value (e.g., 0) if the input string is `null`:
	
	``cpp
	public int getLength(String s) {
	  if (s == null) {
	  return 0;
	  }
	  return s.length();
	}
	``
	
	By adding the null check, you ensure that the method handles the `null` case gracefully, avoiding the `NullPointerException` and providing a more robust implementation."

Comment	The few-shot strategy quickly identifies the null-dereference risk and introduces two clear mitigation options: exception handling or returning a default value. It's practical and implementation-ready but assumes the reader understands the trade-offs between the two.

Strategy:	chain_of_thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Reason step-by-step to find any null-dereference risk in the following Java code:
	``cpp
	public int getLength(String s) {
	  return s.length(); // What if s is null?
	}
	
	``"	

Output:

"To analyze the provided Java code for potential null-dereference risks, we need to follow a step-by-step reasoning process. The code in question is:	To identify any null-dereference risks in the provided Java code, we need to carefully examine the method `getLength(String s)`. Here's a step-by-step analysis:

``java
public int getLength(String s) {
return s.length(); // What if s is null?
}
``
 	
### Step 1: Understand the Method's Purpose 	
The method `getLength` is designed to return the length of a `String` object passed as an argument. It uses the `length()` method of the `String` class.
 	
### Step 2: Identify Potential Null Input 	
The parameter `String s` can potentially be `null`. In Java, if you attempt to call a method on a `null` reference, a `NullPointerException` will be thrown at runtime.
 	
### Step 3: Analyze the Code for Null Handling 	
In the current implementation, there is no check to see whether `s` is `null` before calling `s.length()`. If `s` is `null`, the following line will lead to a `NullPointerException`:
 	
``java
return s.length(); // This line will throw an exception if s is null
``
 	
### Step 4: Consider Scenarios of Method Invocation 	
- If the method is called with a valid `String` (e.g., `""Hello""`), it will return the length correctly (5).
- If the method is called with `null` (e.g., `getLength(null)`), it will throw a `NullPointerException`.
 	
### Step 5: Determine Corrective Measures 	
To mitigate the risk of a null dereference, you can implement a null check at the beginning of the method. Here's an example of how to handle it:
 	
``java
public int getLength(String s) {
if (s == null) {
return 0; // or throw an IllegalArgumentException, or handle it as per requirement
}
return s.length();
}
``
 	
### Step 6: Conclusion 	
In the original code, there is a clear risk of a `NullPointerException` if the input `String s` is `null`. By implementing a null check, you can prevent this exception and handle null inputs gracefully. Thus, it is essential to always consider null inputs when dealing with reference types in Java."
	1. **Method Signature and Return Type**:
	- The method `getLength(String s)` takes a `String` parameter `s` and returns an `int`.
	
	2. **Method Body**:
	- The method returns the result of `s.length()`.
	- `s.length()` is a method call on the `String` object `s`.
	
	3. **Potential Null-Dereference**:
	- If `s` is `null`, calling `s.length()` will result in a `NullPointerException`.
	- This is because attempting to call a method on a `null` reference results in a runtime error.
	
	4. **Risk Assessment**:
	- The method does not check if `s` is `null` before calling `s.length()`.
	- This lack of a null check means that passing a `null` value to `s` will lead to a `NullPointerException`.
	
	### Step-by-Step Solution to Avoid Null-Dereference:
	
	1. **Add a Null Check**:
	- Add a check to see if `s` is `null` before calling `s.length()`.
	- If `s` is `null`, you can either throw an `IllegalArgumentException` or return a special value (e.g., `-1`) to indicate the invalid input.
	
	2. **Updated Method**:
	``java
	public int getLength(String s) {
	if (s == null) {
	throw new IllegalArgumentException("Input string cannot be null");
	// Alternatively, you can return a special value like -1
	// return -1;
	}
	return s.length();
	}
	``
	
	### Explanation:
	- **Null Check**: The `if (s == null)` statement checks if `s` is `null`. If it is, an `IllegalArgumentException` is thrown with an appropriate error message.
	- **Exception Handling**: Throwing an exception is a good practice because it makes the code more robust and clear about the expected input. If you prefer to return a special value, you can do so as shown in the alternative comment.
	
	### Conclusion:
	By adding a null check, the method becomes safer and handles the case where `s` might be `null`, thereby avoiding `NullPointerException`.
	
	``java
	public int getLength(String s) {
	if (s == null) {
	throw new IllegalArgumentException("Input string cannot be null");
	}

	return s.length();
	}
	``
	
	This updated method ensures that `s` is not null before attempting to call `s.length()`, thus mitigating the risk of a null-dereference.

Comment	The chain-of-thought strategy methodically walks through the reasoning behind the null-dereference and its consequences. It's thorough and educational but longer and less direct than needed for fast debugging or implementation.

Model Analysis and Comparison: The few-shot strategy is concise and solution-oriented, making it ideal for quick implementation. The chain-of-thought strategy offers deeper reasoning and better clarity for understanding the underlying issue. While both identify the same risk, the few-shot is more efficient, whereas the chain-of-thought is more instructive.

Task 8 - CSV Parser Variants (Python)

Strategy:	few_shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Example:
	Input: """a,b""",c' → Output: ['a,b','c']
	Now improve the following CSV parser:
	``python
	def parse_csv_line(line):
	  return line.split(',') # Incomplete: doesn't handle quoted fields
	
	``"	

Output:

To improve the CSV parser to handle quoted fields properly, we need to consider that fields enclosed in quotes can contain commas that should not be treated as delimiters. We can achieve this by using Python's built-in `csv` module, which is specifically designed for parsing CSV files and handles quoted fields correctly.
 	
Here's an improved version of the `parse_csv_line` function:
 	
``python
import csv
from io import StringIO
 	
def parse_csv_line(line):
# Use StringIO to treat the string as a file object 	
f = StringIO(line)
reader = csv.reader(f)
# Read the first (and only) row from the CSV reader 	
return next(reader)
 	
# Example usage: 	
	"Certainly! To properly handle quoted fields in a CSV line, you need to account for fields that may contain commas within them. This requires a more sophisticated approach than simply splitting by commas.
	
	Here is an improved version of the CSV parser:
	
	``python
	def parse_csv_line(line):
	  result = []
	  current_field = []
	  in_quotes = False
	
	  for char in line:
	  if char == """:
	  in_quotes = not in_quotes
	  elif char == ',' and not in_quotes:
	  result.append(''.join(current_field))
	  current_field = []
	  else:
	  current_field.append(char)

input_line = '"a,b",c'
output = parse_csv_line(input_line)
print(output) # Output: ['a,b', 'c']
``
 	
### Explanation: 	
1. **StringIO**: We use `StringIO` from the `io` module to create an in-memory file-like object from the string. This allows the `csv.reader` to process it as if it were reading from a file.
2. **csv.reader**: The `csv.reader` takes care of parsing the line correctly, handling quoted fields and commas within those fields.
3. **next(reader)**: Since we are only interested in the first row of the CSV (as we are parsing a single line), we use `next()` to retrieve it.
 	
This implementation will correctly parse CSV lines with quoted fields and handle commas inside quotes appropriately.
	if current_field:
	result.append(''.join(current_field))
	
	return result
	
	# Example usage
	input_line = '"""a,b""",c'
	output = parse_csv_line(input_line)
	print(output) # Output: ['a,b', 'c']
	``
	
	### Explanation:
	1. **Initialization**:
	- `result` is an empty list that will store the final parsed fields.
	- `current_field` is an empty list that will accumulate characters for the current field being parsed.
	- `in_quotes` is a boolean that tracks whether the parser is inside a quoted field.
	
	2. **Iteration through the Line**:
	- For each character in the line:
	- If the character is a double quote (`"`), toggle the `in_quotes` flag.
	- If the character is a comma (`,`) and the parser is not inside quotes, it means the end of the current field. Append the accumulated characters to `result` and reset `current_field`.
	- Otherwise, add the character to `current_field`.
	
	3. **Final Field**:
	- After the loop, if there are any remaining characters in `current_field`, append them to `result`.
	
	This approach ensures that fields enclosed in quotes are correctly handled, even if they contain commas."

Comment	The few-shot strategy provides two alternatives—manual parsing logic and a csv module approach—demonstrating flexibility in implementation. While the manual method shows understanding of control flow, it's less reliable than the cleaner and more robust csv module solution.

Strategy:	chain_of_thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Think step-by-step to split CSV fields correctly, handling quotes, in the following code:
	``python
	def parse_csv_line(line):
	return line.split(',') # Incomplete: doesn't handle quoted fields
	``"	

Output:

"To correctly parse a CSV line while handling quoted fields, we need to account for the fact that commas within quotes should not be treated as delimiters. A simple split on commas will fail in these cases, as it won't recognize when a comma is	"To correctly parse CSV fields while handling quoted fields, you need to account for several scenarios: 1. Quoted fields may contain commas.

part of a quoted string.

A robust way to handle this is to use the `csv` module from Python's standard library, which is specifically designed for parsing CSV files. This module can handle quoted fields, commas inside quotes, and other nuances of CSV formatting.

Here's how you can implement the `parse_csv_line` function using the `csv` module:

```python
import csv
from io import StringIO

def parse_csv_line(line):
    # Use StringIO to simulate a file object for the csv reader
    f = StringIO(line)
    reader = csv.reader(f)
    # Read the first (and only) row from the reader
    return next(reader)

# Example usage
line = 'value1,""value, with comma"",value3'
parsed_fields = parse_csv_line(line)
print(parsed_fields)  # Output: ['value1', 'value, with comma', 'value3']
```

Explanation:
1. **Import the required modules**: Here we import `csv` for parsing CSV content and `StringIO` to create an in-memory file-like object from the string input.

2. **Create a StringIO object**: This allows us to use the string `line` as if it were a file. The `csv.reader` expects an iterable (like a file object) to read from.

3. **Use csv.reader**: We create a CSV reader object, which will automatically handle the complexities of quoted fields, commas within quotes, and other CSV format rules.

4. **Return the parsed line**: We use `next(reader)` to get the first (and in this case, only) row from the reader, which will return a list of fields correctly split.

This approach is clean, efficient, and leverages the built-in capabilities of Python to handle CSV parsing effectively."

2. Fields may be quoted with double quotes (`"""`).
3. Quoted fields may contain escaped quotes, i.e., `""""""` within the field.

You can use Python's built-in `csv` module to handle these complexities. Here is a step-by-step approach using the `csv` module:

1. **Import the `csv` module.**
2. **Use a `StringIO` object to simulate reading from a file.**
3. **Use the `csv.reader` to parse the single line.**

Here's the updated code:

```python
import csv
from io import StringIO

def parse_csv_line(line):
    # Use StringIO to simulate reading from a file
    f = StringIO(line)
    # Create a CSV reader object
    reader = csv.reader(f)
    # Read the line and return the fields
    return next(reader)

# Example usage:
line = 'John,""Doe, Jr."",30,""New York, NY"",50000'
fields = parse_csv_line(line)
print(fields)
```

Explanation:

1. **`import csv`**: Import the `csv` module to handle CSV parsing.
2. **`from io import StringIO`**: Import `StringIO` to simulate reading a single line as a file.
3. **`f = StringIO(line)`**: Create a `StringIO` object with the input line.
4. **`reader = csv.reader(f)`**: Create a CSV reader object that reads from the `StringIO` object.
5. **`next(reader)`**: Use `next` to read the single line from the CSV reader and return the fields.

This approach ensures that quoted fields with commas and escaped quotes are handled correctly.

Example Output:
For the input line `'John,""Doe, Jr."",30,""New York, NY"",50000'`, the output will be:
```
['John', 'Doe, Jr.', '30', 'New York, NY', '50000']
```

This method is robust and leverages Python's built-in capabilities to handle the complexities of CSV parsing."

Comment	The chain-of-thought strategy favors Python's csv module from the start and walks through each step with clear

	justification. It emphasizes robustness and standard library usage, ideal for handling edge cases in real-world CSV parsing.

Model Analysis and Comparison: The few-shot model explores both a custom parser and the standard csv module, offering variety but introducing potential reliability issues with the manual approach. In contrast, the chain-of-thought model consistently promotes the csv module, guiding the user through its implementation with a structured explanation. While both approaches reach a correct and effective solution, the chain-of-thought model is more focused, dependable, and aligned with best practices.

Task 9 - Data Class to API Conversion (Kotlin)

Strategy:	prompt_chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Step 1: Define GET `/product/{id}` returning JSON.
	Step 2: Define POST endpoint accepting a Product in the body for the following code:
	``cpp
	data class Product(val id: Int, val name: String, val price: Double)
	// TODO: Create GET and POST endpoints using Ktor
	
	``"	

Output:

"To create the GET and POST endpoints for the `Product` data class using Ktor in Kotlin, we'll first define the structure of the application and then implement the endpoints. Below is an example of how to achieve this.
 	
### Step 1: Define GET `/product/{id}` returning JSON 	
 	
This endpoint will retrieve a product by its ID.
 	
### Step 2: Define POST endpoint accepting a Product in the body 	
 	
This endpoint will accept a new product in the request body and add it to our list of products.
 	
Here's how you can implement these endpoints using Ktor:
 	
``kotlin
import io.ktor.application.*
import io.ktor.features.ContentNegotiation
import io.ktor.http.HttpStatusCode
import io.ktor.jackson.jackson
import io.ktor.request.receive
import io.ktor.response.respond
import io.ktor.routing.*
import io.ktor.server.engine.embeddedServer
import io.ktor.server.netty.Netty
import io.ktor.serialization.kotlinx.json.*
import kotlinx.serialization.Serializable
import java.util.concurrent.atomic.AtomicInteger
	"To define the GET and POST endpoints for handling `Product` objects using the Ktor framework in Kotlin, you need to follow these steps:
	
	1. **Define the GET `/product/{id}` endpoint returning JSON.**
	2. **Define the POST endpoint accepting a `Product` in the body.**
	
	Below is an example implementation:
	
	### Step 1: Define GET `/product/{id}` returning JSON
	
	First, ensure you have the necessary dependencies in your `build.gradle.kts` file:
	
	``kotlin
	dependencies {
	implementation(""io.ktor:ktor-server-core:2.0.0"")
	implementation(""io.ktor:ktor-server-netty:2.0.0"")
	
	implementation(""io.ktor:ktor-server-content-negotiation:2.0.0"")
	
	implementation(""io.ktor:ktor-serialization-kotlinx-json:2.0.0"")
	implementation(""io.ktor:ktor-server-status-pages:2.0.0"")
	}
	``

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

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

fun main() {
    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            jackson { }
        }
        routing {
            // GET endpoint to retrieve a product by ID
            get(""/product/{id}"") {
                val id = call.parameters[""id""]?.toIntOrNull()
                if (id != null) {
                    val product = products.find { it.id == id }
                    if (product != null) {
                        call.respond(product)
                    } else {
                        call.respond(HttpStatusCode.NotFound, ""Product not
found"")
                    }
                } else {
                    call.respond(HttpStatusCode.BadRequest, ""Invalid ID"")
                }
            }

            // POST endpoint to add a new product
            post(""/product"") {
                val newProduct = call.receive<Product>()
                val productToAdd = newProduct.copy(id =
idCounter.getAndIncrement())
                products.add(productToAdd)
                call.respond(HttpStatusCode.Created, productToAdd)
            }
        }
    }.start(wait = true)
}
```

Explanation of the Code:

1. **Dependencies**: Make sure to include Ktor dependencies in your `build.gradle.kts` file:

   ```kotlin
   implementation(""io.ktor:ktor-server-core:2.x.y"")
   implementation(""io.ktor:ktor-server-netty:2.x.y"")
   implementation(""io.ktor:ktor-serialization-jackson:2.x.y"")
   ```

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

2. **Data Class**: The `Product` class is defined using Kotlin's

Now, create the server application with the GET endpoint:

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

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

fun main() {
    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            json()
        }
        routing {
            get(""/product/{id}"") {
                val id = call.parameters[""id""]?.toIntOrNull()
                val product = Product(id ?: -1, ""Sample Product"",
100.0) // Replace with actual product fetching logic
                if (id != null) {
                    call.respond(product)
                } else {
                    call.respondText(""Product not found"", status =
HttpStatusCode.NotFound)
                }
            }

            post(""/product"") {
                val product = call.receive<Product>()
                // Handle the product, e.g., save to database
                call.respondText(""Product received: $product"",
status = HttpStatusCode.Created)
            }
        }
    }.start(wait = true)
}
```

Step 2: Define POST endpoint accepting a `Product` in the body

The POST endpoint is already defined in the above code. Here is a breakdown of the relevant part:

```kotlin
post(""/product"") {
    val product = call.receive<Product>()
```

`@Serializable` annotation, which allows it to be serialized to and from JSON.
 	
3. **Routing**:
- The **GET** endpoint listens for requests at `/product/{id}`. It retrieves the product by its ID, and responds with either the product data or a 404 error if not found.
- The **POST** endpoint listens for requests at `/product`. It accepts a `Product` object in the body of the request, assigns a new ID to it, adds it to the list of products, and responds with a 201 status and the created product.
 	
4. **In-Memory Storage**: For demonstration purposes, we are using a mutable list to store products and an atomic integer to generate unique IDs.
 	
5. **Running the Server**: The server is started on port 8080, and it will listen indefinitely for incoming requests.
 	
This is a basic implementation and can be expanded with further features like error handling, validation, and integration with a database for persistent storage."
	// Handle the product, e.g., save to database
	call.respondText(""Product received: $product"", status = HttpStatusCode.Created)
	}
	``
	
	### Explanation:
	
	1. **Content Negotiation**: The `ContentNegotiation` feature is installed with `json()` to handle JSON serialization and deserialization.
	2. **GET Endpoint**: The `get(""/product/{id}"")` endpoint retrieves a product by its ID. It responds with a `Product` object in JSON format if the ID is valid.
	3. **POST Endpoint**: The `post(""/product"")` endpoint receives a `Product` object from the request body and responds with a confirmation message.
	
	This setup should allow you to handle GET and POST requests for `Product` objects using the Ktor framework."

Comment	This approach builds the API step by step and delivers a complete, well-structured Ktor application using in-memory storage and unique ID generation. It's comprehensive and scalable but assumes familiarity with advanced Ktor concepts like AtomicInteger and external dependencies.

Strategy:	self_consistency	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Convert the following Kotlin data class to a REST API with GET and POST endpoints using Ktor.
	``cpp
	data class Product(val id: Int, val name: String, val price: Double)
	// TODO: Create GET and POST endpoints using Ktor
	
	``"	

Output:

"Certainly! Below is an example of how you can create a REST API with GET and POST endpoints using Ktor for the provided Kotlin data class `Product`.
 	
First, ensure you have the necessary dependencies in your `build.gradle.kts`:
 	
``kotlin
dependencies {
implementation(""io.ktor:ktor-server-core:2.2.3"")
implementation(""io.ktor:ktor-server-netty:2.2.3"")
 	
implementation(""io.ktor:ktor-server-content-negotiation:2.2.3"")
 	
implementation(""io.ktor:ktor-serialization-kotlinx-json:2.2.3
	"To create a REST API with Ktor for the `Product` data class, you need to set up a Ktor server with `GET` and `POST` endpoints. Below is an example of how you can achieve this:
	
	1. **Add Dependencies**: Make sure you include the necessary dependencies in your `build.gradle.kts` file.
	
	``kotlin
	dependencies {
	implementation(""io.ktor:ktor-server-core:2.0.0"")
	implementation(""io.ktor:ktor-server-netty:2.0.0"")
	implementation(""io.ktor:ktor-server-content-negotiation:2.0.0"")
	implementation(""io.ktor:ktor-serialization-kotlinx-json:2.0.0"")
	implementation(""io.ktor:ktor-server-status-pages:2.0.0"")
	}
	``

```
"")
    implementation(""ch.qos.logback:logback-classic:1.2.11"")
}
```

Next, you can create a simple Ktor application with the required endpoints:

```kotlin
import io.ktor.application.*
import io.ktor.features.ContentNegotiation
import io.ktor.http.HttpStatusCode
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.serialization.kotlinx.json.*
import io.ktor.server.engine.embeddedServer
import io.ktor.server.netty.Netty
import io.ktor.request.*

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

val products = mutableListOf<Product>()

fun main() {
    embeddedServer(Netty, port = 8080, module = Application::module).start(wait = true)
}

fun Application.module() {
    install(ContentNegotiation) {
        json()
    }

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

        post(""/products"") {
            val product = call.receive<Product>()
            products.add(product)
            call.respond(HttpStatusCode.Created, product)
        }
    }
}
```

Explanation:
1. **Data Class**: The `Product` data class is defined as given.
2. **In-memory Storage**: A mutable list `products` is created to store the products in memory.
3. **Main Function**: The `main` function starts an embedded server on port 8080 with the `module` function

2. **Create the Product Data Class**:

```kotlin
import kotlinx.serialization.Serializable

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

3. **Create the REST API with Ktor**:

```kotlin
import io.ktor.application.*
import io.ktor.features.*
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 io.ktor.serialization.*
import io.ktor.statuspages.*

fun main() {
    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            json()
        }
        install(StatusPages) {
            exception<Throwable> { cause ->
                call.respond(HttpStatusCode.InternalServerError, ""500: $cause"")
            }
        }

        routing {
            val products = mutableListOf<Product>()

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

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

Explanation:

defined below.
4. **Routing**:
- The `GET /products` endpoint returns the list of products.
- The `POST /products` endpoint receives a `Product` object, adds it to the list, and responds with the created product and a `201 Created` status.
5. **Content Negotiation**: The Ktor server is configured to use JSON serialization with `kotlinx.serialization`.
 	
### Running the API: 	
To run this application, you can execute the `main` function. You can test the API using tools like Postman or cURL.
 	
- **GET Request**:
``bash
curl http://localhost:8080/products
``
 	
- **POST Request**:
``bash
curl -X POST http://localhost:8080/products -H ""Content-Type: application/json"" -d '{""id"": 1, ""name"": ""Sample Product"", ""price"": 19.99}'
``
 	
This will allow you to create and retrieve products using your Ktor REST API."
	1. **Dependencies**: The necessary dependencies for Ktor, JSON serialization, and content negotiation are included.
	2. **Product Data Class**: The `Product` data class is annotated with `@Serializable` to enable JSON serialization.
	3. **Ktor Server Setup**:
	- The server is set up to run on port 8080.
	- `ContentNegotiation` is installed to handle JSON serialization.
	- `StatusPages` is installed to handle errors gracefully.
	- Routing is defined to handle `GET` and `POST` requests.
	
	4. **Endpoints**:
	- `GET /products`: Returns the list of products.
	- `POST /products`: Receives a product in the request body, adds it to the list, and responds with the created product.
	
	### Running the Server:
	
	To run the server, simply execute the `main` function. The server will start on `http://localhost:8080`, and you can test the endpoints using tools like `curl`, Postman, or any HTTP client.
	
	### Testing the Endpoints:
	
	- **GET `/products`**:
	``sh
	curl -X GET http://localhost:8080/products
	``
	
	- **POST `/products`**:
	``sh
	curl -X POST -H ""Content-Type: application/json"" -d '{""id"": 1, ""name"": ""Sample Product"", ""price"": 19.99}' http://localhost:8080/products
	``
	
	This setup provides a basic REST API for managing a list of products using Ktor. You can expand this further by adding more features like validation, persistence, and authentication as needed."

Comment	The self-consistency strategy sticks closely to the original data class and delivers a clean, functional REST API with essential features like JSON handling and error responses. It's more streamlined and production-ready but slightly less detailed in dynamic behaviors like ID assignment.

Model Analysis and Comparison: The prompt chaining strategy offers a detailed walkthrough, dynamically managing product IDs and explaining setup intricacies, making it ideal for guided learning. The self-consistency strategy is cleaner and more aligned with best practices for maintainable APIs, offering graceful error handling and minimal setup. While both implementations are correct, prompt chaining emphasizes step-by-step completeness, whereas self-consistency focuses on clarity and real-world API readiness.

Task 10 - Function Summarization (Python)

Strategy:	prompt_chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Step 1: Describe what the following function does.
	Step 2: Summarize it in one sentence:
	``python
	def reverse_words(sentence):
	  return ' '.join(sentence.split()[::-1])
	
	``"

Output:

"### Step 1: Describe what the following function does. The `reverse_words` function takes a string input, referred to as `sentence`, splits the string into individual words, reverses the order of those words, and then joins them back together into a single string with spaces in between. The resulting string has the same words as the original but arranged in the opposite order. ### Step 2: Summarize it in one sentence: The `reverse_words` function reverses the order of words in a given sentence."	"### Step 1: Description The function `reverse_words` takes a string `sentence` as input. It performs the following steps: 1. **Splits the Sentence**: The input string is split into a list of words using the `split()` method, which by default splits the string by whitespace. 2. **Reverses the List**: The list of words is then reversed using slicing `[::-1]`. 3. **Joins the List**: The reversed list of words is joined back into a single string with each word separated by a single space using the `' '.join()` method. 4. **Returns the Result**: The final reversed sentence is returned. ### Step 2: Summary The `reverse_words` function reverses the order of words in a given sentence."
Comment	The prompt chaining strategy breaks the task into explicit steps, offering both a detailed explanation and a clean one-sentence summary. It's clear and educational but slightly repetitive between the two steps.

Strategy:	self_consistency	
Model:	gpt-4o-mini	Codestral-2501

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

Output:

The `reverse_words` function takes a string input called `sentence`, splits it into individual words, reverses the order of those words, and then joins them back together into a single string with spaces in between. The resulting string is returned, effectively reversing the word order of the original sentence.	The function `reverse_words` takes a sentence as input and returns a new sentence with the order of the words reversed. It splits the input sentence into words, reverses the list of words, and then joins them back into a single string with spaces in between. This effectively reverses the order of the words in the input sentence.
Comment	The self-consistency strategy generates a concise and accurate explanation directly, avoiding the need for intermediate breakdowns. It's efficient and to the point but less guided for learners who benefit from step-by-step reasoning.

Model Analysis and Comparison: Prompt chaining emphasizes clarity by separating explanation from summary, which helps in educational contexts or when teaching code comprehension. In contrast, the self-consistency strategy delivers a compact, well-phrased summary in one go, making it better suited for documentation or quick reviews. Both are accurate, but prompt chaining offers more structure, while self-consistency offers more brevity.

Task 11 - Prompt from Code Comments (Python)

Strategy:	chain_of_thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Explain step-by-step how to craft a prompt to generate the following code:
	``python
	# 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
	``"	

Output:

"To craft a prompt that generates the provided Python code for checking if a number is prime, follow these steps:
 	
### Step 1: Define the Objective 	
Clearly outline what you want the code to accomplish. In this case, the goal is to create a function that checks if a given number is prime.
 	
### Step 2: Identify Key Features 	
List the essential features you want the function to include:
- The function should accept one parameter (the integer to check).
- It should return a boolean value (True if the number is prime, False otherwise).
- Include logic to handle numbers less than or equal to 1.
- Implement a loop to check for factors up to the square root of the number.
 	
### Step 3: Specify the Programming Language 	
Mention that you want the code written in Python. This ensures that the generated code is in the correct syntax.
 	
### Step 4: Provide Context or Examples (Optional) 	
Providing context or examples can help clarify your request. For instance, you might include that the function should be called `is_prime` and could be tested with a few example numbers.
 	
### Step 5: Write the Prompt 	
Combine all the elements into a coherent prompt. Here's an example:
	"Crafting a prompt to generate the given Python code involves clearly describing the desired functionality and the specific steps involved. Here's a step-by-step guide on how to structure the prompt:
	
	1. **Describe the Overall Functionality**:
	  - Specify that the function is designed to check if a number is prime.
	  - Explain what a prime number is: a natural number greater than 1 that has no positive divisors other than 1 and itself.
	
	2. **Explain the Basic Conditions**:
	  - Numbers less than or equal to 1 are not prime.
	  - If a number `n` is divisible by any number other than 1 and itself, it is not prime.
	
	3. **Detail the Algorithm**:
	  - Start with the base case where numbers less than or equal to 1 are explicitly checked.
	  - Iterate from 2 up to the square root of `n` to check for divisors.
	  - If `n` is divisible by any number in this range, it is not prime.
	  - If no divisors are found, `n` is prime.
	
	4. **Provide the Expected Output**:
	  - The function should return `True` if the number is prime and `False` otherwise.
	
	Here is the structured prompt:
	
	--- 	
	
	Prompt:
	
	Write a Python function named `is_prime` that checks if a given number `n`

``` Generate a Python function named `is_prime` that checks if a given integer is prime. The function should take one parameter, `n`, and return True if `n` is a prime number and False otherwise. The function should handle numbers less than or equal to 1 appropriately and use a loop to check for factors only up to the square root of `n`. ``` ### Step 6: Review and Refine Make sure the prompt is clear, concise, and free of ambiguity. You can also specify the function's name and any other preferences you have for the code style. By following these steps, you create a well-structured prompt that guides the code generation process effectively."	is a prime number. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. **Steps to Implement the Function:** 1. If `n` is less than or equal to 1, return `False` because numbers less than or equal to 1 are not prime. 2. Iterate from 2 up to the square root of `n` (inclusive). This is because a larger factor of `n` must be a multiple of a smaller factor that has already been checked. 3. For each number `i` in this range, if `n` is divisible by `i` (i.e., `n % i == 0`), return `False` because `n` is not prime. 4. If no divisors are found in the loop, return `True` because `n` is prime. **Expected Output:** - `is_prime(2)` should return `True` - `is_prime(4)` should return `False` - `is_prime(19)` should return `True` - `is_prime(1)` should return `False` --- By following this structured approach, you ensure that the prompt provides all necessary details for generating the desired code."
Comment	The chain-of-thought strategy walks through each component of the function and builds a structured prompt grounded in understanding the algorithm. It's instructive and thorough, making it ideal for learners, though slightly verbose for quick prompt crafting.

Strategy:	prompt_chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step 1: Write a system instruction that sets the task.
	Step 2: Provide the comment and ask for implementation for the following code:
	``python
	# 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
	
	``"	

Output:

"### Step 1: System Instruction Your task is to review the provided Python code for a function that checks if a number is prime. Ensure that the implementation is correct, efficient, and adheres to best practices. Suggest any improvements or optimizations if necessary.	"### Step 1: System Instruction **Task Description**: The function `is_prime` is intended to check if a given number `n` is a prime number. The function should return `True` if `n` is prime and `False` otherwise. The function should handle edge cases such as numbers less than or

### Step 2: Comment and Request for Implementation 	
``python
# 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
 	
# The function appears to be correctly implemented for checking primality. 	
# However, to enhance readability and maintainability, consider the following suggestions: 	
# 1. Add a docstring to explain the function's purpose and its parameters. 	
# 2. Use explicit return statements for clarity. 	
# 3. Handle edge cases more explicitly, such as negative numbers. 	
 	
# Please implement these improvements: 	
# - Add a docstring to the `is_prime` function. 	
# - Refactor the function to include more explicit handling of edge cases. 	
``"
	equal to 1.
	
	### Step 2: Comment and Implementation Request
	
	Comment:
	``python
	# 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
	``
	
	Implementation Request:
	1. Ensure the function handles all edge cases correctly.
	2. Optimize the function if possible.
	3. Provide additional comments if necessary.
	4. Test the function with various inputs to verify correctness.
	
	Please implement the function `is_prime` as described."

Comment	The prompt chaining strategy provides a system instruction followed by implementation guidance that emphasizes clarity, docstrings, and explicit handling. It's practical and actionable but assumes the user is already familiar with function correctness.

Model Analysis and Comparison: The chain-of-thought strategy is ideal for users who need step-by-step guidance to reverse-engineer prompts from code, providing deep insight into algorithm logic. The prompt chaining strategy, on the other hand, is more task-focused, directly suggesting improvements and refinements to an existing function. While chain-of-thought aids in understanding, prompt chaining better supports real-world refactoring and documentation needs.

Task 12 - Fixing Factorial Bug (Python)

Strategy:	self-consistency	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"The logic fails for n = 0 due to incorrect loop bounds. Revise it so the function returns 1 for 0 and behaves consistently for all inputs.
	``python
	def factorial(n):
	  result = 1
	  for i in range(1, n):
	  result *= i
	  return result
	``"	

Output:

"The issue in the provided `factorial` function is that it does not correctly handle the case when \(n = 0 \) and also has incorrect loop bounds for computing the factorial. The factorial of 0 is defined to be 1. Additionally, the loop should run up to and including \(n \) to calculate the factorial correctly for positive integers.	"The current implementation of the `factorial` function fails for `n = 0` because the loop does not execute any iterations, leaving `result` as 1, which is incorrect. Instead, the factorial of 0 is

Here's a revised version of the `factorial` function that handles the case for \(n = 0 \) and calculates the factorial correctly for all non-negative integers:
 	
``python
def factorial(n):
  if n < 0:
  raise ValueError(""Factorial is not defined for negative numbers."")
  result = 1
  for i in range(1, n + 1):
  result *= i
  return result
``
 	
### Explanation: 	
- **Base Case**: The function checks if \(n \) is less than 0 and raises a `ValueError` since factorial is not defined for negative numbers.
- **Loop Bounds**: The loop runs from 1 to \(n \) (inclusive) using `range(1, n + 1)`, ensuring that all numbers up to \(n \) are multiplied.
- **Return Value**: The function returns 1 when \(n = 0 \) because the loop does not execute, and `result` remains 1, which is the correct value for \(0! \).
 	
### Testing the Function: 	
You can test the function with different values of \(n \) to ensure it behaves as expected:
 	
``python
print(factorial(0)) # Output: 1
print(factorial(1)) # Output: 1
print(factorial(5)) # Output: 120
print(factorial(3)) # Output: 6
``"
	defined as 1.
	
	You can revise the function to handle the special case of `n = 0` explicitly and adjust the loop bounds to ensure correctness for other values of `n`.
	
	Here is the revised code:
	
	``python
	def factorial(n):
	  if n == 0:
	  return 1
	  result = 1
	  for i in range(1, n + 1):
	  result *= i
	  return result
	``
	
	In this revised version:
	
	1. The function checks if `n` is 0 and returns 1 immediately if true.
	2. For other values of `n`, the loop iterates from 1 to `n` inclusive, correctly multiplying the `result` by each integer from 1 to `n`.
	
	This ensures that the function behaves consistently for all non-negative integer inputs."

Comment	GPT-4o-mini correctly identifies and fixes the loop bound and adds error handling for negative inputs, providing a robust and well-documented solution. Codestral also handles the base case and adjusts the loop but lacks the additional check for negative numbers, making it slightly less defensive.

Strategy:	chain-of-thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step-by-step, identify how the loop behaves for n = 0 and adjust it to include the base case explicitly.
	``python
	def factorial(n):
	  result = 1
	  for i in range(1, n):
	  result *= i
	  return result
	``"	

Output:

"Let's analyze the provided code step-by-step and see how it behaves when `n = 0`. Then, we'll adjust the code to handle the	"To understand how the loop behaves for \(n = 0 \) and to adjust it to include the base case explicitly, let's break it down step-by-step.

base case explicitly.

Step-by-Step Analysis of the Loop for `n = 0`

1. **Function Call**: When we call `factorial(0)`, the function is executed with `n` set to 0.

2. **Initialization**: Inside the function, `result` is initialized to 1.
   ```python
   result = 1
   ```

3. **Loop Execution**: The for loop is defined as:
   ```python
   for i in range(1, n):
   ```
   - Here, `range(1, n)` generates a sequence of numbers starting from 1 up to (but not including) `n`.
   - Since `n` is 0, `range(1, 0)` results in an empty sequence. Therefore, the loop body is never executed.

4. **Return Statement**: The function reaches the return statement:
   ```python
   return result
   ```
   - Since the loop did not execute, `result` remains 1.

5. **Return Value**: The function returns 1, which is not the correct factorial value for 0. The factorial of 0 is defined to be 1, but we need to ensure our function handles this explicitly.

Adjusting the Code to Include the Base Case

To correctly handle the base case for `n = 0`, we can add an explicit check at the beginning of the function. Here's the adjusted code:

```python
def factorial(n):
    # Base case: if n is 0, return 1
    if n == 0:
        return 1

    result = 1
    for i in range(1, n + 1):  # Note: changed to n + 1 to include n itself
        result *= i
    return result
```

Key Changes Made:

1. **Base Case Check**: We added a condition to return 1 immediately if `n` is 0.
   ```python

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

Behavior for \(n = 0 \)
1. **Initialization**: `result` is initialized to 1.
2. **Loop Range**: `range(1, 0)` generates an empty range, so the loop does not execute.
3. **Return**: The function returns `result`, which is still 1.

This implies that the original function returns 1 for \(n = 0 \), which is incorrect because the factorial of 0 is defined as 1.

Adjustments
To fix the issue, we need to:
1. Handle the base case explicitly.
2. Ensure the loop correctly iterates from 1 to \(n \) (inclusive).

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

Explanation
1. **Base Case**: The function first checks if \(n \) is 0. If it is, the function returns 1.
2. **Loop Adjustment**: The loop now iterates from 1 to \(n \) inclusively using `range(1, n + 1)`.
3. **Return**: The function returns the correctly calculated factorial.

Testing
Let's test the function with a few values:
- `factorial(0)` should return 1.
- `factorial(1)` should return 1.
- `factorial(2)` should return 2.
- `factorial(3)` should return 6.
- `factorial(4)` should return 24.
- `factorial(5)` should return 120.

Final Code
```python
def factorial(n):
    if n == 0:

| | |
|---|---|
| if n == 0:<br>  return 1<br>```<br><br>2. **Loop Adjustment**: We adjusted the loop to iterate from 1 to `n + 1`, so that it includes `n` in the multiplication.<br>```python<br>for i in range(1, n + 1):<br>```<br><br>### Final Behavior for `n = 0`<br><br>With these changes:<br>- When `factorial(0)` is called, it will hit the base case and return 1, which is the correct output.<br><br>The function can now correctly compute the factorial for any non-negative integer, including 0." | return 1<br>result = 1<br>for i in range(1, n + 1):<br>  result *= i<br>return result<br><br># Testing the function<br>print(factorial(0))  # Should print 1<br>print(factorial(1))  # Should print 1<br>print(factorial(2))  # Should print 2<br>print(factorial(3))  # Should print 6<br>print(factorial(4))  # Should print 24<br>print(factorial(5))  # Should print 120<br>```<br><br>This ensures that the function correctly handles the base case and computes the factorial for all non-negative integers." |
| Comment | GPT-4o-mini thoroughly walks through the logic, clearly explaining the edge case for n = 0 and how the fix ensures consistent behavior across inputs.<br>Codestral mirrors this step-by-step structure, offering a similar correction, but adds explicit test cases that reinforce functional correctness for multiple inputs. |

**Model Analysis and Comparison:** Both GPT-4o-mini and Codestral-2501 correctly revise the factorial function, but GPT-4o-mini includes additional safeguards such as error handling for negative inputs, showing a deeper understanding of edge case coverage. GPT-4o-mini tends to be slightly more verbose and explanatory, while Codestral is more concise and focused on implementation. Overall, GPT-4o-mini demonstrates a more thorough approach under both self-consistency and chain-of-thought prompting, giving it a slight edge in robustness and clarity.

**Task 13 - Linked List Node Deletion (C)**

| Strategy: | few-shot | |
|---|---|---|
| Model: | gpt-4o-mini | Codestral-2501 |
| Prompt: | "Example:<br>Input: struct Node *head; int key = 4;<br>Output: Node with value 4 is removed from the list.<br><br>Now remove the node with the given key from the linked list.<br>```cpp<br>struct Node {<br>  int data;<br>  struct Node *next;<br>};<br>void deleteNode(struct Node **head, int key) {<br>  // TODO : Implement node deletion<br>}<br>```" | |

**Output:**

"To implement the function `deleteNode` that removes a node with a given key from a linked list, you need to consider several cases:

1. The list is empty.
2. The node to be deleted is the head node.
3. The node to be deleted is somewhere in the middle or end of the list.

Here's how you can implement the `deleteNode` function:

```cpp
#include <iostream>

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

void deleteNode(struct Node **head, int key) {
    // Check if the list is empty
    if (*head == nullptr) {
        std::cout << ""List is empty, nothing to delete."" << std::endl;
        return;
    }

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

    // If the head node itself holds the key to be deleted
    if (temp != nullptr && temp->data == key) {
        *head = temp->next; // Change head
        delete temp; // Free old head
        std::cout << ""Node with value "" << key << "" is removed from the list."" << std::endl;
        return;
    }

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

    // If the key was not present in the linked list
    if (temp == nullptr) {
        std::cout << ""Node with value "" << key << "" not found in the list."" << std::endl;
        return;
    }

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

    delete temp; // Free memory
    std::cout << ""Node with value "" << key << "" is removed from the list."" << std::endl;
}
```

"To implement the node deletion function in a linked list, you need to handle different scenarios such as when the node to be deleted is the head of the list, in the middle, or the end of the list. Here is the complete implementation of the `deleteNode` function:

```cpp
#include <iostream>

// Definition for singly-linked list node
struct Node {
    int data;
    struct Node *next;
};

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

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Changed head
        delete temp;        // Free old head
        std::cout << ""Node with value "" << key << "" is removed from the list."" << std::endl;
        return;
    }

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

    // If key was not present in the linked list
    if (temp == NULL) {
        std::cout << ""Node with value "" << key << "" not found in the list."" << std::endl;
        return;
    }

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

    delete temp; // Free memory

    std::cout << ""Node with value "" << key << "" is removed from the list."" << std::endl;
}

// Utility function to print the linked list
void printList(struct Node *node) {
    while (node != NULL) {
        std::cout << node->data << "" "";
```

```
// Function to insert a new node at the end of the list for testing purposes
void insertNode(struct Node **head, int newData) {
    struct Node *newNode = new Node();
    newNode->data = newData;
    newNode->next = nullptr;

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

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

// Function to print the linked list for testing purposes
void printList(struct Node *node) {
    while (node != nullptr) {
        std::cout << node->data << "" "";
        node = node->next;
    }
    std::cout << std::endl;
}

int main() {
    struct Node *head = nullptr;

    // Create a linked list for testing: 1 -> 2 -> 3 -> 4
    insertNode(&head, 1);
    insertNode(&head, 2);
    insertNode(&head, 3);
    insertNode(&head, 4);

    std::cout << ""Initial list: "";
    printList(head);

    int key = 3; // Key to be deleted
    deleteNode(&head, key);

    std::cout << ""Updated list after deleting "" << key << "": "";
    printList(head);

    return 0;
}
```

Explanation:
1. The function first checks if the list is empty. If it is, it prints a message and returns.
2. It checks if the head node contains the key. If it does, it updates the head to point to the next node and deletes the old head.

```
        node = node->next;
    }
    std::cout << std::endl;
}

int main() {
    // Creating a simple linked list: 1 -> 2 -> 3 -> 4 -> 5
    struct Node* head = new Node();
    head->data = 1;
    head->next = new Node();
    head->next->data = 2;
    head->next->next = new Node();
    head->next->next->data = 3;
    head->next->next->next = new Node();
    head->next->next->next->data = 4;
    head->next->next->next->next = new Node();
    head->next->next->next->next->data = 5;
    head->next->next->next->next->next = NULL;

    std::cout << ""Original list: "";
    printList(head);

    // Delete node with key = 4
    deleteNode(&head, 4);

    std::cout << ""List after deletion: "";
    printList(head);

    // Clean up the allocated memory
    while (head != NULL) {
        struct Node *temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

Explanation:
1. **Node Structure**: The `Node` structure represents a node in the linked list with an integer `data` and a pointer `next` to the next node.

2. **deleteNode Function**:
   - **Head Node Deletion**: If the head node itself holds the key to be deleted, update the head to the next node and delete the old head.
   - **Search for Key**: Traverse the list to find the node with the given key while keeping track of the previous node.
   - **Key Not Found**: If the key is not found in the list, print a message indicating the key was not found.
   - **Delete Node**: If the key is found, unlink the node from the list and delete it.

3. If the node to be deleted is not the head, it traverses the list to find the node, keeping track of the previous node. 4. Once the node is found, it unlinks the node from the list and frees its memory. 5. If the key is not found, it prints a message indicating that the node does not exist. This code is ready to run and can be tested with different values for the linked list and the key to be deleted."	3. **Utility Functions**: - `printList`: To print the linked list for verification. - `main`: To create a simple linked list, delete a node, and print the list before and after deletion. This implementation ensures that the linked list is correctly modified and any removed nodes are properly deallocated to avoid memory leaks."
Comment	GPT-4o-mini delivers a complete implementation with robust handling of edge cases, including empty lists and head node deletion, and adds helper functions for testing. Codestral's version is similarly correct and efficient, with a slightly more compact structure, but it doesn't include helper functions for node insertion, making it less test-friendly out-of-the-box.

Strategy:	chain-of-thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step-by-step, walk through identifying the target node and safely unlinking it.
	``cpp
	struct Node {
	int data;
	struct Node *next;
	};
	void deleteNode(struct Node **head, int key) {
	// TODO : Implement node deletion
	}
	``"	

Output:

"To implement the `deleteNode` function that safely unlinks a node from a linked list, we need to follow a series of steps. The goal is to find the target node based on the specified key, unlink it from the list, and free the memory associated with it. Here's a step-by-step breakdown of how to do this: ### Step 1: Define the function signature We already have the function signature. The function accepts a pointer to the head of the linked list and an integer key that identifies the node to be deleted. ### Step 2: Handle edge cases Before we start traversing the list, we need to check if the linked list is empty or if the head is `NULL`. If the list is empty, we can simply return since there's nothing to delete. ### Step 3: Initialize pointers We need to maintain two pointers: - `prev`: to keep track of the previous node as we traverse the list. - `current`: to iterate through the nodes.	"Sure, let's walk through the process of identifying the target node and safely unlinking it from the singly linked list in the `deleteNode` function. We'll do this step-by-step: ### Step-by-Step Implementation 1. **Check if the list is empty**: - If the list is empty (i.e., `head` is `NULL`), there's nothing to delete. 2. **Check if the node to be deleted is the head node**: - If the node to be deleted is the head node, update the head to the next node. 3. **Traverse the list to find the node to be deleted**: - Use a `temp` pointer to traverse the list and a `prev` pointer to keep track of the previous node. 4. **Find the node with the specified key**: - Traverse the list until you find the node with the `key` or reach the end of the list. 5. **Unlink the node**:

Step 4: Traverse the list
Iterate through the linked list, looking for the node whose data matches the key. During this traversal, we need to update both `prev` and `current` pointers.

Step 5: Unlink the node
Once we find the node to delete:
- If it's the head node, we need to update the head pointer.
- If it's not the head, we simply adjust the `next` pointer of the previous node to skip the current node.
- Finally, free the memory allocated for the node.

Step 6: Implement the function
Here is how the complete implementation looks:

```cpp
#include <iostream>

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

void deleteNode(struct Node **head, int key) {
    // Step 2: Check if the list is empty
    if (*head == NULL) {
        std::cout << ""List is empty."" << std::endl;
        return;
    }

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

    // Step 4: Traverse the list
    while (current != NULL && current->data != key) {
        prev = current;
        current = current->next;
    }

    // If the key was not found in the list
    if (current == NULL) {
        std::cout << ""Node with key "" << key << "" not found."" << std::endl;
        return;
    }

    // Step 5: Unlink the node
    if (prev == NULL) {
        // The node to be deleted is the head
        *head = current->next; // Change head
    } else {
        // Node to be deleted is not the head
        prev->next = current->next; // Unlink the node
    }
```

- If the node is found, update the `next` pointer of the previous node to skip the node to be deleted.

6. **Free the memory of the deleted node**:
   - Use `free` to release the memory of the deleted node.

Implementation

Here's the complete implementation:

```cpp
#include <iostream>

// Definition of the Node structure
struct Node {
    int data;
    struct Node *next;
};

// Function to delete a node with a specific key
void deleteNode(struct Node **head, int key) {
    // Step 1: Check if the list is empty
    if (*head == NULL) {
        std::cout << ""The list is empty."" << std::endl;
        return;
    }

    // Step 2: Check if the node to be deleted is the head node
    struct Node *temp = *head;
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Change head
        free(temp);         // Free old head
        return;
    }

    // Step 3: Traverse the list to find the node to be deleted
    struct Node *prev = NULL;
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // Step 4: If the key was not present in the linked list
    if (temp == NULL) {
        std::cout << ""Key not found in the list."" << std::endl;
        return;
    }

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

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

```
    // Free memory
    delete current;
    std::cout << ""Node with key "" << key << "" deleted."" <<
std::endl;
}
```

Explanation of the Code
1. **Edge Case Check**: We check if the linked list is empty (`*head == NULL`). If it is, we print a message and return.
2. **Traversal**: We use a `while` loop to traverse the linked list. We move `prev` to `current` and `current` to `current->next` until we either find the node with the matching key or reach the end of the list (`current == NULL`).
3. **Node Deletion**: If `current` becomes `NULL`, we print that the node was not found. Otherwise, we check if `prev` is `NULL`, indicating that we need to delete the head node. We adjust the pointers accordingly to unlink the node from the list.
4. **Memory Management**: Finally, we free the memory used by the node being deleted to avoid memory leaks.

This function can be used to delete a node from a singly linked list safely and effectively."

```
// Function to print the linked list (for testing purposes)
void printList(struct Node *head) {
    struct Node *temp = head;
    while (temp != NULL) {
        std::cout << temp->data << "" "";
        temp = temp->next;
    }
    std::cout << std::endl;
}

// Function to push a new node to the front of the list (for testing purposes)
void push(struct Node **head, int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}

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

    // Push some nodes to the list
    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);

    std::cout << ""Original list: "";
    printList(head);

    // Delete a node
    deleteNode(&head, 1);
    std::cout << ""List after deleting 1: "";
    printList(head);

    // Delete another node
    deleteNode(&head, 3);
    std::cout << ""List after deleting 3: "";
    printList(head);

    return 0;
}
```

Explanation

1. **Initialization and Edge Cases**:
   - We check if the list is empty and if the node to be deleted is the head node.

2. **Traversal**:
   - We traverse the list using a `temp` pointer and keep track of the previous node using a `prev` pointer.

	3. **Finding and Unlinking the Node**: - We find the node with the specified `key` and unlink it by updating the `next` pointer of the previous node. 4. **Memory Management**: - We free the memory of the deleted node to prevent memory leaks. This implementation ensures that the node is safely unlinked from the list and the memory is properly managed."
Comment	GPT-4o-mini provides a step-by-step breakdown of the logic, clearly explaining traversal, deletion, and memory management, making the thought process transparent and instructive. Codestral takes a slightly more procedural approach, explaining steps as implementation guidelines, and includes both memory management and a working test example with dynamic list construction.

Model Analysis and Comparison: Both GPT-4o-mini and Codestral-2501 deliver correct and thorough implementations of linked list node deletion using both few-shot and chain-of-thought prompting strategies. GPT-4o-mini tends to offer more modular, testable code with clear helper functions and step-by-step logic ideal for learning and debugging. Codestral, meanwhile, emphasizes concise and structured execution, making it efficient but slightly less pedagogical than GPT-4o-mini in scenarios that benefit from additional contextual scaffolding.

Task 14 - Recursive Function Completion (Python)

Strategy:	zero-shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Complete the recursive Fibonacci function to return the nth number.
	``python
	def fibonacci(n):
	  # TODO : Base cases and recursive call
	  pass
	``"	

Output:

"To complete the recursive Fibonacci function, you need to define the base cases and the recursive call. The Fibonacci sequence is defined as follows:
 	
- `fibonacci(0) = 0`
- `fibonacci(1) = 1`
- For `n > 1`, `fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)`
 	
Here's how you can implement this in Python:
 	
``python
def fibonacci(n):
  # Base cases
  if n == 0:
  return 0
  elif n == 1:
  return 1
  else:
	"Sure! To complete the recursive Fibonacci function, you need to handle the base cases and implement the recursive call. The Fibonacci sequence is defined as follows:
	
	- \(F(0) = 0 \)
	- \(F(1) = 1 \)
	- \(F(n) = F(n-1) + F(n-2) \) for \(n > 1 \)
	
	Here's the complete implementation:
	
	``python
	def fibonacci(n):
	  # Base cases
	  if n == 0:
	  return 0
	  elif n == 1:
	  return 1
	  # Recursive call
	  else:

# Recursive call return fibonacci(n - 1) + fibonacci(n - 2) # Example usage print(fibonacci(10)) # Output: 55 ``` In this implementation: - The function returns `0` if `n` is `0`. - It returns `1` if `n` is `1`. - For all other values of `n`, it calculates the Fibonacci number by summing the results of the two preceding numbers in the sequence recursively."	return fibonacci(n-1) + fibonacci(n-2) # Example usage print(fibonacci(10)) # Output: 55 ``` This function correctly implements the recursive definition of the Fibonacci sequence. It handles the base cases where \(n \) is 0 or 1 and uses the recursive case for \(n > 1 \). However, note that this implementation is not efficient for large values of \(n \) due to redundant calculations. For better performance, consider using memoization or an iterative approach."
Comment	GPT-4o-mini and Codestral both correctly implemented the recursive Fibonacci function, covering base cases and the recursive structure. However, only Codestral explicitly warns about performance inefficiencies due to repeated computations, suggesting the potential use of memoization.

Strategy:	few-shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Example:
	Input: fibonacci(0) → 0
	Input: fibonacci(1) → 1
	Output: fibonacci(5) → 5
	
	Now implement the recursive function:
	``python
	def fibonacci(n):
	# TODO : Base cases and recursive call
	pass
	``"	

Output:

"Sure! The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The base cases for the Fibonacci function are defined as follows:
 	
- `fibonacci(0)` returns `0`
- `fibonacci(1)` returns `1`
 	
For any other positive integer `n`, the function can be defined recursively as:
 	
- `fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)`
 	
Here's the implementation of the recursive Fibonacci function in Python:
 	
``python
def fibonacci(n):
	"Certainly! The Fibonacci sequence is a series of numbers where each number (after the first two) is the sum of the two preceding ones. The sequence typically starts with 0 and 1. For example, the sequence starts as follows: 0, 1, 1, 2, 3, 5, 8, ...
	
	To implement the Fibonacci function recursively, you need to define the base cases and the recursive case. Here is the implementation:
	
	``python
	def fibonacci(n):
	# Base cases
	if n == 0:
	return 0
	elif n == 1:
	return 1
	else:
	# Recursive call
	return fibonacci(n - 1) + fibonacci(n - 2)

# Base cases 	
if n == 0:
return 0
elif n == 1:
return 1
else:
# Recursive call 	
return fibonacci(n - 1) + fibonacci(n - 2)
 	
# Example usage 	
print(fibonacci(0)) # Output: 0
print(fibonacci(1)) # Output: 1
print(fibonacci(5)) # Output: 5
``
 	
You can use this function to calculate the Fibonacci number for any non-negative integer `n`."
	# Example usage
	print(fibonacci(5)) # Output: 5
	``
	
	### Explanation:
	1. **Base Cases**:
	- If `n` is 0, the function returns 0.
	- If `n` is 1, the function returns 1.
	
	2. **Recursive Case**:
	- For any other value of `n`, the function returns the sum of the two preceding Fibonacci numbers, which are computed by recursive calls to `fibonacci(n - 1)` and `fibonacci(n - 2)`.
	
	### Note:
	This recursive implementation is straightforward but can be inefficient for large values of `n` due to repeated calculations. For better performance, consider using memoization or an iterative approach. However, this implementation serves to demonstrate the basic recursive Fibonacci function."

Comment	Both models effectively mirrored the provided examples and delivered the correct recursive Fibonacci implementation. Codestral stood out slightly for including a clearer explanation of how the sequence works and cautioning about efficiency concerns.

Model Analysis and Comparison: Both GPT-4o-mini and Codestral-2501 produce correct implementations of the recursive Fibonacci function using zero-shot and few-shot strategies. While GPT-4o-mini focuses on clarity and functional correctness, Codestral goes a step further by providing insight into performance limitations and potential optimizations. Across both strategies, Codestral slightly edges ahead for its practical awareness and better contextual framing of recursion efficiency.

Task 15 - Constructor Completion (Python)

Strategy:	zero-shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Write a constructor that initializes `name`, `age`, and optional `email` fields.
	``python
	class Person:
	def __init__(self):
	# TODO : Add name , age , and optional email
	pass
	``"	

Output:

"You can create a constructor for the `Person` class that initializes the `name`, `age`, and an optional `email` field as follows:
 	
``python
class Person:
	"To initialize the `name`, `age`, and optional `email` fields in the `Person` class, you can modify the constructor to accept parameters for these fields. You can set the `email` field to `None` by default to make it optional.
	
	Here is how you can do it:

def __init__(self, name, age, email=None):
self.name = name # Required field
self.age = age # Required field
self.email = email # Optional field, default is None
``
 	
With this constructor, you can create a `Person` object by providing values for `name` and `age`, while `email` can be omitted. Here are some examples of how to use this class:
 	
``python
# Creating a Person with all fields 	
person1 = Person(""Alice"", 30, ""alice@example.com"")
 	
# Creating a Person without an email 	
person2 = Person(""Bob"", 25)
 	
# Accessing the attributes 	
print(person1.name) # Output: Alice
print(person1.age) # Output: 30
print(person1.email) # Output: alice@example.com
 	
print(person2.name) # Output: Bob
print(person2.age) # Output: 25
print(person2.email) # Output: None
``
 	
This implementation allows for flexibility in creating `Person` objects with or without the `email` field."
	``python
	class Person:
	def __init__(self, name, age, email=None):
	self.name = name
	self.age = age
	self.email = email
	
	# Example usage:
	person1 = Person(""Alice"", 30)
	person2 = Person(""Bob"", 25, ""bob@example.com"")
	
	print(person1.name, person1.age, person1.email) # Output: Alice 30 None
	print(person2.name, person2.age, person2.email) # Output: Bob 25 bob@example.com
	``
	
	In this code:
	- The `__init__` method takes `name` and `age` as required parameters.
	- The `email` parameter is optional and defaults to `None` if not provided.
	- The `self` parameter refers to the instance being created and allows you to set instance attributes."

Comment	Both models correctly implemented a constructor with required name and age attributes and an optional email, defaulting to None. GPT-4o-mini includes a broader explanation and demonstration of usage, whereas Codestral focuses on a concise, direct example.

Strategy:	prompt chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step 1: Identify which attributes should be stored.
	Step 2: Use parameters in `__init__` to initialize them.
	``python
	class Person:
	def __init__(self):
	# TODO : Add name , age , and optional email
	pass
	``"	

Output:

"To store the attributes `name`, `age`, and an optional `email` in the `Person` class, you can modify the `__init__` method to accept these parameters. Here's how you can do it: ```python	"To complete the `Person` class so that it can store the attributes `name`, `age`, and an optional `email`, you can follow these steps: 1. Identify the attributes: `name`, `age`, and `email`. 2. Use parameters in the `__init__` method to initialize these attributes.

class Person:
  def __init__(self, name, age, email=None):
  self.name = name # Required attribute
  self.age = age # Required attribute
  self.email = email # Optional attribute, defaults to None if not provided
 	
# Example of creating a Person instance 	
person1 = Person(name=""Alice"", age=30)
person2 = Person(name=""Bob"", age=25, email=""bob@example.com"")
 	
print(person1.name, person1.age, person1.email) # Output: Alice 30 None
print(person2.name, person2.age, person2.email) # Output: Bob 25 bob@example.com
``
 	
In this implementation:
 	
1. The `__init__` method takes three parameters: `name`, `age`, and an optional `email`.
2. The `name` and `age` parameters are required, while `email` is optional and defaults to `None` if not provided.
3. The attributes are stored in the instance using `self`."
	Here's how you can implement it:
	
	``python
	class Person:
	  def __init__(self, name, age, email=None):
	  self.name = name
	  self.age = age
	  self.email = email
	
	# Example usage:
	p1 = Person(""Alice"", 30)
	p2 = Person(""Bob"", 25, ""bob@example.com"")
	
	print(p1.name) # Output: Alice
	print(p1.age) # Output: 30
	print(p1.email) # Output: None
	
	print(p2.name) # Output: Bob
	print(p2.age) # Output: 25
	print(p2.email) # Output: bob@example.com
	``
	
	In this implementation:
	- The `__init__` method takes three parameters: `name`, `age`, and `email` (with a default value of `None`).
	- The `self.name`, `self.age`, and `self.email` attributes are initialized based on the provided arguments."

Comment	GPT-4o-mini offers a clean, structured breakdown of each implementation step with inline comments and usage examples. Codestral delivers a similarly correct version but explains the logic in a slightly more procedural format with clear distinctions between required and optional attributes.

Model Analysis and Comparison: Both GPT-4o-mini and Codestral-2501 produced accurate and idiomatic Python constructors using zero-shot and prompt chaining strategies. GPT-4o-mini tends to provide more explanatory and user-guided responses, making it especially helpful for beginners or instructional contexts. Codestral, while slightly more concise, maintains clarity and correctness, offering a streamlined experience with minimal verbosity.

Task 16 - Binary Search Completion (Java)

Strategy:	few-shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Example:
	Input: binarySearch([1, 3, 5, 7], 5) → 2
	Now implement binary search to find the target in the array.
	``java
	public int binarySearch(int[] arr, int target) {
	  int left = 0, right = arr.length - 1;
	  while (left <= right) {
	  int mid = (left + right) / 2;
	  // TODO : Compare and adjust bounds
	  }	

	return -1;
	}
	``"

Output:

"To implement the binary search algorithm in Java, we need to compare the target value with the value at the midpoint of the current search range. If the target is equal to the midpoint value, we return the index of the midpoint. If the target is less than the midpoint value, we adjust the right boundary to search in the left half of the array. If the target is greater, we adjust the left boundary to search in the right half. This process continues until we either find the target or exhaust the search range.
 	
Here's the complete implementation:
 	
``java
public int binarySearch(int[] arr, int target) {
int left = 0, right = arr.length - 1;
 	
while (left <= right) {
int mid = (left + right) / 2; // Calculate the midpoint
 	
// Compare the target with the value at mid
if (arr[mid] == target) {
return mid; // Target found, return the 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 the Code: 	
1. **Initialization**: We initialize two pointers, `left` and `right`, to indicate the current search range.
2. **Loop until the range is exhausted**: The loop continues as long as `left` is less than or equal to `right`.
3. **Calculate the midpoint**: The midpoint `mid` is calculated as the average of `left` and `right`.
4. **Compare the target**:
- If `arr[mid]` equals the target, we return `mid`.
- If `arr[mid]` is less than the target, we adjust `left` to `mid + 1` to search the right half.
- If `arr[mid]` is greater than the target, we adjust `right` to `mid - 1` to search the left half.
5. **Return -1 if not found**: If the loop finishes without finding the target, we return -1 to indicate that the target is not present in the array.
	"Certainly! To implement the binary search algorithm, you need to compare the middle element of the array with the target value and adjust the bounds (`left` and `right`) accordingly. If the middle element is equal to the target, you return the middle index. If the target is less than the middle element, you adjust the right bound; if the target is greater, you adjust the left bound.
	
	Here's the complete implementation of the `binarySearch` method:
	
	``java
	public int binarySearch(int[] arr, int target) {
	int left = 0, right = arr.length - 1;
	while (left <= right) {
	int mid = (left + right) / 2;
	
	// Check if the target is present at mid
	if (arr[mid] == target) {
	return mid;
	}
	
	// If target is greater, ignore the left half
	if (arr[mid] < target) {
	left = mid + 1;
	}
	// If target is smaller, ignore the right half
	else {
	right = mid - 1;
	}
	}
	// Target is not present in the array
	return -1;
	}
	``
	
	### Explanation:
	1. **Initialization**:
	- `left` is initialized to the first index of the array (0).
	- `right` is initialized to the last index of the array (`arr.length - 1`).
	
	2. **Loop Condition**:
	- The loop continues as long as `left` is less than or equal to `right`.
	
	3. **Calculate Middle Index**:
	- `mid` is calculated as the average of `left` and `right`.
	
	4. **Comparison**:
	- If `arr[mid]` equals `target`, the method returns `mid`.
	- If `arr[mid]` is less than `target`, it means the target is in the right half, so `left` is updated to `mid + 1`.
	- If `arr[mid]` is greater than `target`, it means the target is in the left half, so `right` is updated to `mid - 1`.

### Example Usage: 	
``java
public static void main(String[] args) {
int[] arr = {1, 3, 5, 7};
int target = 5;
 	
int result = binarySearch(arr, target);
System.out.println(""Index of target "" + target + "": "" + result); // Output: Index of target 5: 2
}
``
 	
This code will allow you to use the `binarySearch` method to find the index of a target value in a sorted array."
	5. **Return -1**:
	- If the loop exits without finding the target, it returns -1 indicating that the target is not in the array.
	
	This implementation ensures that the array is searched efficiently, with a time complexity of \(O(\log n)\)."

Comment	Both GPT-4o-mini and Codestral-2501 correctly implement the binary search algorithm with appropriate bound adjustments and clear condition checks. GPT-4o-mini offers slightly more context and includes a runnable main method for demonstration, while Codestral is slightly more concise but equally correct and thorough.

Strategy:	chain-of-thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step-by-step, adjust the left/right bounds depending on how mid compares to the target.
	``java
	public int binarySearch(int[] arr, int target) {
	int left = 0, right = arr.length - 1;
	while (left <= right) {
	int mid = (left + right) / 2;
	// TODO : Compare and adjust bounds
	}
	return -1;
	}
	``"	

Output:

"To implement the binary search algorithm, you need to compare the value at the `mid` index with the `target`. Depending on whether the value at `mid` is less than, greater than, or equal to the `target`, you'll adjust the `left` and `right` bounds accordingly. Here's how you can complete the `binarySearch` method step-by-step: 1. **Compare the value at `mid` with `target`:** - If the value at `mid` is equal to `target`, you return `mid` as the index where the target is found. - If the value at `mid` is less than `target`, you need to search the right half of the array, so you adjust the `left` bound to `mid + 1`. - If the value at `mid` is greater than `target`, you need to search the left half of the array, so you adjust the `right` bound to `mid - 1`.	"To complete the `binarySearch` method, you need to compare the value at the `mid` index to the `target` and adjust the `left` and `right` bounds accordingly. Here's a step-by-step approach to achieve this: 1. **Compare `mid` with `target`:** - If `arr[mid]` is equal to `target`, return `mid` since you have found the target. - If `arr[mid]` is less than `target`, adjust the `left` bound to `mid + 1` to search the right half. - If `arr[mid]` is greater than `target`, adjust the `right` bound to `mid - 1` to search the left half. 2. **Loop until `left` exceeds `right`:** - The loop should continue as long as `left` is less than or equal to `right`. 3. **Return `-1` if the target is not found:** - If the loop exits without finding the target, return `-1`.

2. **Continue the loop until you either find the target or the bounds converge (i.e., `left` exceeds `right`).**
 	
Here's the completed code:
 	
``java
public int binarySearch(int[] arr, int target) {
  int left = 0, right = arr.length - 1;
  while (left <= right) {
  int mid = (left + right) / 2;
 	
  // Compare and adjust bounds
  if (arr[mid] == target) {
  return mid; // Target found
  } else if (arr[mid] < target) {
  left = mid + 1; // Search in the right half
  } else {
  right = mid - 1; // Search in the left half
  }
  }
 	
  return -1; // Target not found
}
``
 	
### Explanation of the Code: 	
- The `while` loop continues as long as `left` is less than or equal to `right`.
- Inside the loop, `mid` is calculated as the midpoint between `left` and `right`.
- The comparison checks if the element at `mid` is equal to the `target`. If it is, `mid` is returned.
- If the element at `mid` is less than the `target`, it means the target must be in the right half of the array, so we adjust `left`.
- Conversely, if the element at `mid` is greater than the `target`, it means the target must be in the left half, so we adjust `right`.
- If the target is not found after the loop finishes, `-1` is returned to indicate that the target is not in the array."
	Here is the complete code with these steps implemented:
	
	``java
	public int binarySearch(int[] arr, int target) {
	  int left = 0;
	  int right = arr.length - 1;
	
	  while (left <= right) {
	  int mid = left + (right - left) / 2; // To avoid potential overflow
	
	  if (arr[mid] == target) {
	  return mid;
	  } else if (arr[mid] < target) {
	  left = mid + 1; // Search right half
	  } else {
	  right = mid - 1; // Search left half
	  }
	  }
	
	  return -1; // Target not found
	}
	``
	
	### Explanation:
	- **Mid Calculation:** Using `left + (right - left) / 2` instead of `(left + right) / 2` to avoid potential overflow issues.
	- **Comparison:** Compare `arr[mid]` with `target`.
	  - If they are equal, return `mid`.
	  - If `arr[mid]` is less than `target`, update `left` to `mid + 1`.
	  - If `arr[mid]` is greater than `target`, update `right` to `mid - 1`.
	- **Loop Condition:** The loop continues as long as `left` is less than or equal to `right`.
	- **Return `-1`:** If the loop exits without finding the target, return `-1`.
	
	This approach ensures that you efficiently search for the target in a sorted array using binary search."

Comment	GPT-4o-mini provides a well-structured walkthrough of the algorithm's logic with code and explanation neatly paired. Codestral mirrors the same logic, with an added note about using left + (right - left) / 2 to avoid overflow—showing attention to edge-case safety.

Model Analysis and Comparison: GPT-4o-mini and Codestral-2501 both deliver functionally correct and logically sound binary search implementations using few-shot and chain-of-thought strategies. While GPT-4o-mini focuses on clarity and completeness with usage examples, Codestral provides a slightly more technical and optimized solution by highlighting integer overflow prevention. In chain-of-thought mode especially, Codestral edges out with a more defensive coding practice, making it a stronger pick for production-level robustness.

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

Strategy:	self-consistency

Model:	gpt-4o-mini	Codestral-2501

Prompt:	"The function is named isOdd but returns true for even numbers. Fix the logic or rename the function to match its behavior.
	``cpp
	bool isOdd(int x) {
	  return x % 2 == 0;
	}
	``"	

Output:

"The function `isOdd` is incorrectly implemented to return true for even numbers. To fix the logic, you should modify the return statement to check for odd numbers instead. Here's the corrected version of the function:
 	
``cpp
bool isOdd(int x) {
  return x % 2 != 0; // Return true if x is odd
}
``
 	
Alternatively, if you want to keep the function name but change its behavior to match its name, you could rename the function to something more appropriate, like `isEven`:
 	
``cpp
bool isEven(int x) {
  return x % 2 == 0; // Return true if x is even
}
``
 	
Choose the option that best fits your needs!"
	"The provided function `isOdd` returns `true` for even numbers because it checks if `x % 2 == 0`, which is true for even numbers. To fix the logic or rename the function to match its behavior, you can either adjust the logic to check for odd numbers or rename the function to reflect its current behavior.
	
	### Fix the Logic
	If you intended the function to check for odd numbers, you should modify the condition:
	
	``cpp
	bool isOdd(int x) {
	  return x % 2 != 0; // Check if the number is odd
	}
	``
	
	### Rename the Function
	If you prefer to keep the current logic of checking for even numbers, you should rename the function to reflect that:
	
	``cpp
	bool isEven(int x) {
	  return x % 2 == 0; // Check if the number is even
	}
	``
	
	### Choose the Right Option
	If the intention is to check for odd numbers, go with the first option. If the intention is to check for even numbers, go with the second option.
	
	Here is the corrected version based on the assumption that the function should check for odd numbers:
	
	``cpp
	bool isOdd(int x) {
	  return x % 2 != 0; // Check if the number is odd
	}
	``
	
	And here is the renamed version based on the assumption that the function should check for even numbers:
	
	``cpp
	bool isEven(int x) {
	  return x % 2 == 0; // Check if the number is even

	} ```"
Comment	Both models accurately identify the mismatch between the function's name and logic, and propose either correcting the logic or renaming the function. GPT-4o-mini emphasizes flexibility in fixing the issue, while Codestral gives a slightly clearer distinction between choosing logic correction vs. renaming.

Strategy:	chain-of-thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step-by-step, identify the contradiction between the function's intent and logic, then revise accordingly.
	``cpp
	bool isOdd(int x) {
	  return x % 2 == 0;
	}
	``"	

Output:

"The provided function `isOdd` is intended to determine whether a given integer `x` is odd. However, the implementation contains a logical contradiction. Let's break this down step-by-step:
 	
### Step 1: Identify the Function's Intent 	
The intended purpose of the function `isOdd` is to return `true` if the integer `x` is odd and `false` if it is even.
 	
### Step 2: Analyze the Logic 	
The current implementation of the function is:
``cpp
return x % 2 == 0;
``
This line checks if `x` is even (`x % 2 == 0`). If `x` is even, it returns `true`, which contradicts the function name `isOdd`. Therefore, if `x` is odd, the function should return `false`, but instead, it incorrectly returns `false` when it should return `true`.
 	
### Step 3: Identify the Contradiction 	
- Function Name: `isOdd` suggests it should return `true` for odd numbers.
- Logic: The function currently returns `true` for even numbers and `false` for odd numbers.
 	
### Step 4: Revise the Function 	
To align the implementation with the intended purpose, we need to change the logic to correctly return `true` when `x` is odd.
 	
### Revised Function 	
``cpp
bool isOdd(int x) {
	"Let's break down the function step by step to identify the contradiction between its intent and logic, and then revise it accordingly.
	
	### Step-by-Step Analysis:
	
	1. **Function Declaration:**
	  ``cpp
	  bool isOdd(int x) {
	  ``
	  The function is named `isOdd`, which suggests it should return `true` if the input `x` is odd and `false` if it is even.
	
	2. **Function Logic:**
	  ``cpp
	  return x % 2 == 0;
	  ``
	  The logic inside the function checks if `x % 2` is equal to `0`, which means `x` is even.
	
	### Contradiction:
	
	The function name `isOdd` indicates that the function should determine if a number is odd, but the logic checks if a number is even.
	
	### Revision:
	
	To correct the function, we need to modify the logic to check if `x` is odd instead of even.
	
	``cpp
	bool isOdd(int x) {
	  return x % 2 != 0;
	}
	``

return x % 2 != 0; // Return true if x is odd
}
``
 	
### Explanation of the Revised Function 	
- `x % 2 != 0` checks if `x` is not divisible by 2, which means `x` is odd.
- The function will now correctly return `true` for odd integers and `false` for even integers.
 	
### Final Code 	
Here is the corrected version of the function:
``cpp
bool isOdd(int x) {
return x % 2 != 0; // Correctly checks for odd numbers
}
``"
	Or, equivalently, we can negate the condition to check for an odd number:
	
	``cpp
	bool isOdd(int x) {
	return !(x % 2 == 0);
	}
	``
	
	### Final Revised Function:
	
	``cpp
	bool isOdd(int x) {
	return x % 2 != 0;
	}
	``
	
	This revision aligns the function's intent (to check if a number is odd) with its logic (to check if `x % 2` is not equal to `0`)."

Comment	GPT-4o-mini walks through the contradiction step-by-step, then clearly revises the function to match the intended behavior. Codestral does the same but adds an alternative using logical negation (!(x % 2 == 0)), showing deeper consideration for equivalent expressions.

Model Analysis and Comparison: GPT-4o-mini and Codestral-2501 both successfully resolve the naming and logic conflict in the isOdd function across self-consistency and chain-of-thought strategies. GPT-4o-mini offers user-friendly explanations and guidance for choosing between renaming or rewriting, while Codestral slightly outshines in the chain-of-thought strategy by providing multiple equivalent logical expressions. Overall, both models perform well, but Codestral displays a marginal advantage in showcasing alternative valid implementations.

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

Strategy:	zero-shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Fix the function so it returns a Boolean value when checking for even numbers.
	``cpp
	function isEven(n) {
	return n % 2; // Returns 1 or 0 , not true / false
	}
	``"	

Output:

"To modify the function so that it returns a Boolean value when checking for even numbers, you can update the return statement to explicitly check if `n % 2` equals zero. Here's the corrected function:
 	
``cpp
function isEven(n) {
return n % 2 === 0; // Returns true if n is even, false otherwise
	"To fix the function so it returns a Boolean value (either `true` or `false`) when checking for even numbers, you can compare the result of the modulo operation to `0`. In JavaScript, the expression `n % 2 == 0` will evaluate to `true` if `n` is even and `false` otherwise.
	
	Here is the corrected function:
	
	``cpp
	function isEven(n) {

}
``
 	
In this revised function, `n % 2 === 0` will evaluate to `true` if `n` is even (i.e., when `n` is divisible by 2 with no remainder) and `false` otherwise."
	return n % 2 === 0; // Returns true if n is even, false otherwise
	}
	``
	
	This way, the function will return `true` for even numbers and `false` for odd numbers."

Comment	Both models correctly fix the function to return a Boolean value using the strict equality check n % 2 === 0. GPT-4o-mini offers a slightly more detailed explanation of why this correction is necessary, while Codestral is more direct and to the point.

Strategy:	prompt chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step 1: Identify what `n % 2` returns.
	Step 2: Modify the function so the result is explicitly a Boolean.
	``cpp
	function isEven(n) {
	return n % 2; // Returns 1 or 0 , not true / false
	}
	``"	

Output:

"To modify the function `isEven` so that it explicitly returns a Boolean value (`true` or `false`), we can check the result of `n % 2` and return `true` when `n` is even (i.e., when `n % 2` equals `0`), and `false` when it's odd (i.e., when `n % 2` equals `1`).
 	
Here's the modified function:
 	
``cpp
function isEven(n) {
return n % 2 === 0; // Returns true if n is even, false if n is odd
}
``
 	
### Explanation of the Changes: 	
- We replaced the return statement `return n % 2;` with `return n % 2 === 0;`.
- The expression `n % 2 === 0` evaluates to `true` if `n` is even (when the remainder of `n` divided by `2` is `0`), and `false` otherwise (when the remainder is `1`).
- This ensures that the function explicitly returns a Boolean value.
 	
### Example Usage: 	
``cpp
console.log(isEven(4)); // true
console.log(isEven(5)); // false
``"
	"To modify the `isEven` function so that it explicitly returns a Boolean value, you need to convert the result of `n % 2` to a Boolean. In JavaScript, `0` is falsy and any non-zero value is truthy, so you can directly use the expression `n % 2` in a conditional context to get a Boolean value.
	
	Here is the modified function:
	
	``cpp
	function isEven(n) {
	return n % 2 === 0; // Returns true if n is even, false if n is odd
	}
	``
	
	### Explanation:
	1. **Identify what `n % 2` returns:**
	- `n % 2` returns `0` if `n` is even.
	- `n % 2` returns `1` if `n` is odd.
	
	2. **Modify the function to return a Boolean:**
	- Use the expression `n % 2 === 0` to explicitly check if `n` is even.
	- `n % 2 === 0` will return `true` if `n` is even.
	- `n % 2 === 0` will return `false` if `n` is odd.
	
	This modification ensures that the function `isEven` returns a Boolean value indicating whether the input number `n` is even or not."

Comment	GPT-4o-mini walks through each step clearly, offering concise logic and example usage that clarifies the fix. Codestral does the same but adds contextual insight into JavaScript's truthy/falsy behavior, showing a deeper

	understanding of language semantics.

Model Analysis and Comparison: Both GPT-4o-mini and Codestral-2501 correctly identify the bug and provide a precise fix in both zero-shot and prompt chaining modes. GPT-4o-mini focuses more on clarity and guided reasoning, making it approachable for learners, while Codestral includes stronger contextual awareness of JavaScript behavior, such as truthy/falsy evaluation. Overall, Codestral slightly outperforms in depth of explanation during prompt chaining, although both models are effective and accurate.

Task 19 - Summary Decomposition (C++)

Strategy:	chain-of-thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Step-by-step, explain what the function checks and how it transforms the input.
	``cpp
	int process(int x) {
	  if (x < 0) return -1;
	  return x * x;
	}
	``"	

Output:

"Let's break down the function `process(int x)` step by step to understand what it checks and how it transforms the input.
 	
### Step 1: Function Definition 	
The function is defined as:
``cpp
int process(int x)
``
This means that the function named `process` takes a single integer argument `x` and returns an integer.
 	
### Step 2: Check if Input is Negative 	
The first operation in the function is:
``cpp
if (x < 0) return -1;
``
- **What it checks**: This line checks if the value of `x` is less than 0 (i.e., it checks if `x` is a negative number).
- **What it returns**: If the condition is true (meaning `x` is negative), the function returns `-1`. This serves as an indication that the input was invalid or out of the expected range (since negative numbers are not processed further).
 	
### Step 3: Calculate the Square of the Input 	
The next part of the function is:
``cpp
return x * x;
``
- **What it does**: If `x` is not negative (i.e., `x` is either zero or a positive integer), this line calculates the square of `x` by multiplying `x` by itself (`x * x`).
- **What it returns**: The result of this multiplication (which is `x` squared) is then returned as the output of the function.
	"The function `process` in the provided C++ code takes an integer `x` as input and performs a series of checks and transformations based on the value of `x`. Here is a step-by-step explanation of what the function does:
	
	1. **Input Parameter:**
	  - The function `process` takes an integer `x` as its input.
	
	2. **Check if `x` is Negative:**
	  - The function first checks if `x` is less than 0. This is done using the conditional statement `if (x < 0)`.
	  - If the condition `x < 0` is true (i.e., `x` is negative), the function returns `-1` immediately.
	  - **Transformation 1:** If `x` is negative, the function outputs `-1`.
	
	3. **Square the Input:**
	  - If the condition `x < 0` is false (i.e., `x` is zero or positive), the function proceeds to the next line.
	  - The function then calculates the square of `x` using the expression `x * x`.
	  - **Transformation 2:** If `x` is zero or positive, the function outputs `x * x`.
	
	4. **Return the Result:**
	  - The result of the calculation (either `-1` or `x * x`) is returned as the output of the function.

### Summary of Input Transformation 	
- If `x` is a negative number, the function returns `-1`.
- If `x` is zero or a positive number, the function returns the square of `x`.
 	
### Example Cases 	
1. **Input: -5**
  - Check: `-5 < 0` is true.
  - Output: `-1`
 	
2. **Input: 0**
  - Check: `0 < 0` is false.
  - Calculation: `0 * 0 = 0`
  - Output: `0`
 	
3. **Input: 3**
  - Check: `3 < 0` is false.
  - Calculation: `3 * 3 = 9`
  - Output: `9`
 	
### Conclusion 	
The function `process` checks whether the input integer `x` is negative. If it is, it returns `-1`. If `x` is non-negative, it returns the square of `x`. This function effectively validates the input and transforms it by squaring it when valid."
	### Summary of Transformations:
	- **Input Transformation 1:** If the input `x` is negative, the output is `-1`.
	- **Input Transformation 2:** If the input `x` is zero or positive, the output is `x * x`.
	
	### Example Scenarios:
	
	1. **If `x` is negative (e.g., `x = -5`):**
	  - The function checks if `-5 < 0`, which is true.
	  - The function returns `-1`.
	
	2. **If `x` is zero (e.g., `x = 0`):**
	  - The function checks if `0 < 0`, which is false.
	  - The function returns `0 * 0`, which is `0`.
	
	3. **If `x` is positive (e.g., `x = 4`):**
	  - The function checks if `4 < 0`, which is false.
	  - The function returns `4 * 4`, which is `16`.
	
	By following these steps, the function ensures that the output is always a non-negative integer."

Comment	GPT-4o-mini provides a detailed breakdown with examples and a clean summary, demonstrating clear logical flow. Codestral-2501 presents similar content but with slightly more formal structure and fewer illustrative cases.

Strategy:	prompt chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step 1: Validate input.
	Step 2: Describe the square operation and why it's used.
	``cpp
	int process(int x) {
	  if (x < 0) return -1;
	  return x * x;
	}
	``"	

Output:

"Let's break down the process of validating input and explaining the square operation in the given C++ code snippet. ### Step 1: Validate Input In the provided code, input validation is done by checking if the input integer `x` is less than zero. If `x` is negative, the function returns `-1`, indicating that the input is invalid. This approach ensures that only non-negative integers are processed, as squaring a negative number might not be	"Certainly! Let's break down the provided C++ function `process` step by step and provide additional context. ### Step 1: Validate Input The function `process` first validates its input to ensure it's non-negative. If the input `x` is less than 0, it returns `-1` to indicate an invalid input. This is a common practice to handle unexpected or erroneous inputs gracefully. ### Step 2: Describe the Square Operation and Why It's Used

desired in certain contexts (e.g., mathematical operations where only non-negative results are meaningful).

Here's the relevant part of the code that performs the validation:
```cpp
if (x < 0) return -1;
```
This line checks if `x` is less than zero. If it is, the function returns `-1` to signal an error or invalid input.

Step 2: Describe the Square Operation and Why It's Used
The square operation in the code is represented by the line:
```cpp
return x * x;
```
Squaring a number means multiplying it by itself. The square operation is commonly used in various mathematical applications, including geometry, statistics, and physics. Some reasons for using the square operation include:

1. **Area Calculation**: In geometry, the area of a square (a shape with equal sides) is calculated using the square of the length of one of its sides. Thus, squaring is essential when calculating areas.

2. **Mathematical Properties**: Squaring a number can help simplify various mathematical expressions and equations. For example, it is often used in quadratic equations.

3. **Distance Measurement**: In the context of the Cartesian coordinate system, the square operation is used in the calculation of the distance between points. The distance formula involves the square of the differences in coordinates.

4. **Statistical Analysis**: In statistics, squaring differences from the mean is used to calculate variance, which measures data dispersion.

In the context of the function `process`, squaring a non-negative integer is a straightforward operation that produces a non-negative result. This aligns with the initial validation step that ensures only non-negative integers are processed. The function effectively computes the square of the input while handling invalid inputs appropriately.

Complete Function Summary
Here's a summary of the function `process` and its overall purpose:
```cpp
int process(int x) {
    if (x < 0) return -1; // Input validation: return -1 for invalid input
    return x * x;         // Compute and return the square of x
}
```

The function then proceeds to compute the square of the input `x`. The square operation is defined as `x * x`, which means multiplying the number by itself. This operation is commonly used for several reasons:

1. **Mathematical Analysis**: Squaring numbers is a fundamental operation in mathematics. It is used in various mathematical contexts, such as finding areas, volumes, and in algebraic manipulations.

2. **Statistical Analysis**: In statistics, squaring deviations from the mean is used in the calculation of variance and standard deviation.

3. **Programming and Algorithms**: Squaring a number is often used in algorithms and data structures, such as in the Euclidean distance formula, sorting algorithms, and more.

4. **Physical Models**: In physics and engineering, squaring terms is used to model certain behaviors, such as energy calculations in mechanics.

The Function Code

Here is the provided function with additional comments for clarity:

```cpp
int process(int x) {
    // Step 1: Validate input
    if (x < 0) return -1; // Return -1 for invalid input (x < 0)

    // Step 2: Describe the square operation and why it's used
    // Compute the square of x
    return x * x; // Return the square of x
}
```

Explanation:

1. **Validation**: The function checks if `x` is less than 0. If so, it returns `-1` to indicate an invalid input.
2. **Square Operation**: If `x` is non-negative, the function computes `x * x` and returns the result.

Example Usage:

```cpp
#include <iostream>

int main() {
    int result;

    result = process(4); // Valid input, should return 16
    std::cout << ""Result for 4: "" << result << std::endl;

    result = process(-3); // Invalid input, should return -1
    std::cout << ""Result for -3: "" << result << std::endl;
```

``
The function ensures that only valid (non-negative) integers are squared, providing a reliable output while handling errors gracefully."
	return 0;
	}
	``
	
	### Output:
	
	``
	Result for 4: 16
	Result for -3: -1
	``
	
	In summary, the `process` function is designed to validate input and compute the square of a non-negative integer, highlighting the importance of the square operation in various mathematical and computational contexts."

Comment	GPT-4o-mini emphasizes practical applications of squaring and clearly separates validation and transformation. Codestral-2501 adds broader mathematical and statistical contexts, making its explanation more academic in tone.

Model Analysis and Comparison: GPT-4o-mini excels at giving clear, concise, and example-driven explanations, which can be easier to follow. Codestral-2501, on the other hand, often provides more formal and exhaustive context, appealing to technically advanced readers. While both are accurate, GPT tends to favor accessibility, whereas Codestral leans toward thoroughness.

Task 20 - Purpose Inference → Completion (Python)

Strategy:	few-shot	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Example:
	Input: [90, 80, 70] → Output: 80.0
	Now complete the function to return the average score.
	``python
	def calculate_average(scores):
	  total = 0
	  # TODO : Complete to return average
	  pass
	``"	

Output:

"To complete the `calculate_average` function to return the average score from the provided list of scores, you need to calculate the sum of the scores and then divide by the number of scores. Here's how you can implement it:
 	
``python
def calculate_average(scores):
  if not scores: # Check if the list is empty
  return 0.0 # Return 0.0 for an empty list to avoid division by zero
 	
  total = sum(scores) # Calculate the sum of the scores
	"To complete the function `calculate_average` so that it returns the average score from a list of scores, you can follow these steps:
	
	1. Sum all the scores in the list.
	2. Divide the total sum by the number of scores to get the average.
	3. Return the calculated average.
	
	Here is the complete function:
	
	``python
	def calculate_average(scores):
	  if not scores: # Check if the list is empty

average = total / len(scores) # Calculate the average
return average # Return the average score
 	
# Example usage 	
print(calculate_average([90, 80, 70])) # Output: 80.0
``
 	
### Explanation: 	
1. **Check for an empty list**: Before performing calculations, we check if the `scores` list is empty to avoid division by zero.
2. **Calculate the sum**: We use Python's built-in `sum()` function to get the total of all scores.
3. **Calculate the average**: We divide the total by the number of scores, which is obtained using `len(scores)`.
4. **Return the average**: Finally, we return the calculated average score.
 	
This function will work correctly for both non-empty lists and empty lists."
	return 0.0 # Return 0.0 if the list is empty to avoid division by zero
	
	total = sum(scores) # Sum all the scores
	count = len(scores) # Get the number of scores
	average = total / count # Calculate the average
	return average
	
	# Example usage:
	scores = [90, 80, 70]
	print(calculate_average(scores)) # Output: 80.0
	``
	
	In this implementation:
	- We first check if the `scores` list is empty to avoid division by zero.
	- We then use the `sum` function to get the total sum of the scores.
	- The `len` function gives the number of scores.
	- The average is calculated by dividing the total sum by the number of scores.
	- Finally, the function returns the calculated average."

Comment	Both models return correct and concise solutions, effectively handling edge cases like empty lists. GPT-4o-mini is slightly more streamlined, while Codestral-2501 provides more structured narration.

Strategy:	chain-of-thought	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Step-by-step, explain how to sum the elements and divide by the count for the average.
	``python
	def calculate_average(scores):
	total = 0
	# TODO : Complete to return average
	pass
	``"	

Output:

"To calculate the average of a list of scores in Python, you can follow these steps:
 	
1. **Initialize a total variable**: This will hold the cumulative sum of the scores as you iterate through the list.
 	
2. **Count the number of scores**: You can use the built-in `len()` function to get the total number of scores in the list.
 	
3. **Sum the scores**: Loop through each score in the list, adding each score to the total.
 	
4. **Calculate the average**: Divide the total sum by the count of scores. Make sure to handle the case where there might be no scores (to avoid division by zero).
	"To calculate the average of a list of scores, you need to follow these steps:
	
	1. **Sum the elements** of the list.
	2. **Count the number of elements** in the list.
	3. **Divide the total sum** by the count to get the average.
	
	Here's a step-by-step explanation of how to complete the `calculate_average` function in Python:
	
	``python
	def calculate_average(scores):
	# Step 1: Sum the elements
	total = sum(scores) # This built-in function sums all elements in the list
	
	# Step 2: Count the number of elements
	count = len(scores) # This built-in function returns the number of

5. **Return the average**: Finally, return the calculated average.
 	
Here's how you can implement this in the given function:
 	
``python
def calculate_average(scores):
  total = 0
  count = len(scores) # Count the number of scores
 	
  # Sum up all the scores
  for score in scores:
  total += score
 	
  # Check if count is greater than 0 to avoid division by zero
  if count > 0:
  average = total / count # Calculate average
  else:
  average = 0 # If there are no scores, return 0 or handle it as needed
 	
  return average # Return the calculated average
``
 	
### Explanation of the Code: 	
- **Initialization**: `total` is initialized to `0` to start the accumulation of scores.
- **Counting**: `count` is calculated using `len(scores)` which gives the number of elements in the `scores` list.
- **Summation Loop**: The `for` loop iterates over each score in the `scores` list, adding each score to `total`.
- **Average Calculation**: After the loop, if `count` is greater than `0`, the average is calculated by dividing `total` by `count`. If there are no scores, it assigns `0` to `average`.
- **Return Statement**: Finally, the function returns the calculated average value.
 	
This function will now correctly compute the average of any list of numerical scores passed to it."
	elements in the list
	
	  # Step 3: Divide the total sum by the count to get the average
	  if count == 0:
	  return 0 # To handle the case where the list is empty
	  average = total / count
	
	  return average
	``
	
	Here's a breakdown of the function:
	
	1. **Sum the elements**: Use the `sum()` function to add up all the elements in the `scores` list.
	2. **Count the number of elements**: Use the `len()` function to get the number of elements in the `scores` list.
	3. **Divide the total sum by the count**: Ensure that the count is not zero to avoid division by zero. If the count is zero, return 0 to handle the case of an empty list. Otherwise, compute the average by dividing the total by the count.
	
	Let's add some example usage to see how it works:
	
	``python
	# Example usage
	scores = [85, 90, 92, 88, 95]
	average = calculate_average(scores)
	print(f""The average score is: {average}"")
	``
	
	In this example, the `scores` list contains 5 elements, and the average will be calculated as follows:
	- Total sum: \(85 + 90 + 92 + 88 + 95 = 450\)
	- Count: 5
	- Average: \(\frac{450}{5} = 90 \)
	
	So the output will be:
	``
	The average score is: 90.0
	``
	
	This function handles the basic case of calculating the average and includes a check for an empty list."

Comment	GPT-4o-mini demonstrates detailed reasoning through manual summation, reflecting deeper step-by-step logic. Codestral-2501 favors clarity and efficiency by combining explanation with built-in functions.

Model Analysis and Comparison: GPT-4o-mini generally favors more compact and efficient solutions, especially when prompted step-by-step. Codestral-2501 leans into readability and pedagogy, making its outputs beginner-friendly. While both models are accurate, GPT shows stronger algorithmic reasoning, whereas Codestral excels in clarity and teaching-style explanation.

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

Strategy:	prompt chaining	
Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Step 1: Read the CSV file safely.
	Step 2: Summarize the column.
	Step 3: Identify areas to improve structure or error handling.
	``python
	# utils.py
	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:]]
	  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()
	``"	

Output:

"The provided Python code reads a CSV file, summarizes a specified column by calculating the total and average values, and prints the results. However, there are areas in the code that could be improved, especially concerning error handling and structure. Here's a breakdown of improvements:
 	
### Step 1: Read the CSV File Safely 	
The current implementation of `read_csv` lacks error handling, which can lead to unhandled exceptions if the file does not exist, if it's not accessible, or if there are issues with the CSV format.
 	
### Step 2: Summarize the Column 	
The `summarize_column` function assumes that the data can be converted to `float`. If there are non-numeric values in the specified column, this will raise a `ValueError`. Also, there's no check for an empty dataset or for the specified index being out of range.
 	
### Step 3: Identify Areas to Improve Structure or Error Handling 	
1. **Error Handling**: Add try-except blocks to handle
	"Your provided Python script is a good starting point for reading a CSV file and summarizing a specific column. However, there are several areas for improvement related to structure, error handling, and efficiency. Below, I'll highlight these areas and suggest enhancements:
	
	### Areas to Improve
	
	1. **Error Handling**:
	  - Handle file not found errors.
	  - Handle non-numeric data in the column to be summarized.
	  - Handle empty files or files with inappropriate formats.
	
	2. **Code Structure**:
	  - Separate concerns by creating functions with single responsibilities.
	  - Use constants or configuration for file paths.
	
	3. **Efficiency**:
	  - Use a generator for reading the CSV file to handle large files efficiently.
	  - Validate and sanitize data before processing.
	
	### Enhanced Version

potential errors when opening the file and converting data types.
2. **Input Validation**: Validate the index provided for the column to prevent index errors.
3. **CSV Header Management**: Ensure that the header is processed properly and that the user is informed if the dataset is empty.
4. **Returning More Informative Errors**: Make error messages clearer to help users understand what went wrong.

Here's a revised version of the code that addresses these issues:

```python
# utils.py
import csv
import os

def read_csv(filepath):
    if not os.path.isfile(filepath):
        raise FileNotFoundError(f""The file {filepath} does not exist."")

    with open(filepath, 'r') as f:
        reader = csv.reader(f)
        data = list(reader)

    if not data:
        raise ValueError(""The CSV file is empty."")

    return data

def summarize_column(data, index):
    if index < 0 or index >= len(data[0]):
        raise IndexError(""Column index is out of bounds."")

    values = []
    for row in data[1:]:  # Skip header
        try:
            values.append(float(row[index]))
        except ValueError as e:
            print(f""Warning: Could not convert value '{row[index]}' to float. Skipping this value."")

    if not values:
        raise ValueError(""No valid numeric entries found in the specified column."")

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

def main():
    filepath = 'data.csv'
    try:
        data = read_csv(filepath)
```

```python
# utils.py
import csv
import os

def read_csv(filepath):
    if not os.path.exists(filepath):
        raise FileNotFoundError(f""The file {filepath} does not exist."")

    def read_file():
        with open(filepath, 'r') as f:
            reader = csv.reader(f)
            header = next(reader)  # Read the header
            for row in reader:
                yield row

    return list(read_file())

def summarize_column(data, index):
    try:
        values = [float(row[index]) for row in data]
    except ValueError as e:
        raise ValueError(f""Non-numeric data found in column {index}. Error: {e}"")

    if not values:
        raise ValueError(""No numeric data to summarize."")

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

def main():
    filepath = 'data.csv'
    try:
        data = read_csv(filepath)
        if not data:
            print(""The file is empty."")
            return

        total, avg = summarize_column(data, 1)
        print(""Total:"", total)
        print(""Average:"", avg)
    except FileNotFoundError as fnf_error:
        print(fnf_error)
    except ValueError as val_error:
        print(val_error)
    except Exception as ex:
        print(f""An unexpected error occurred: {ex}"")

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

total, avg = summarize_column(data, 1) # Assuming we want to summarize the second column
print(""Total:"", total)
print(""Average:"", avg)
except Exception as e:
print(f""Error: {e}"")
 	
if __name__ == '__main__':
main()
``
 	
### Key Changes Made: 	
1. **File Existence Check**: Before attempting to open the file, check if it exists.
2. **Empty File Check**: Raise an error if the file is empty.
3. **Column Index Validation**: Check if the provided index is valid for the data.
4. **Value Conversion Handling**: Use a try-except block to catch conversion errors and provide warnings for non-numeric values while processing.
5. **General Exception Handling**: Catch and print any errors that occur during the execution of `main()`.
 	
These changes will make the code more robust and user-friendly, as it can now handle various edge cases more gracefully."
	### Explanation of Changes
	
	1. **Error Handling**:
	- `read_csv` now checks if the file exists and raises a `FileNotFoundError` if it doesn't.
	- `summarize_column` handles `ValueError` to catch non-numeric data and raises a `ValueError` with a descriptive message.
	- `main` catches and prints errors related to file operations and data processing.
	
	2. **Code Structure**:
	- `read_csv` now yields rows one at a time using a generator and checks for the file's existence.
	- The main logic in `main` is encapsulated in a try-except block to handle errors gracefully.
	
	3. **Efficiency**:
	- The generator in `read_csv` efficiently handles large files by reading and processing one row at a time.
	- The script checks for the presence of data before attempting to summarize it, avoiding unnecessary operations.
	
	These changes should make the script more robust, easier to maintain, and better equipped to handle various edge cases."

Comment	GPT's output demonstrates strong structural improvements with detailed error handling and informative messages, leading to more user-friendly and maintainable code. Codestral's version is equally robust but places more emphasis on efficiency and modularity, using generators and clearer exception messages.

Strategy:	self-consistency	
Model:	gpt-4o-mini	Codestral-2501

Prompt	"Ensure the logic is robust when parsing input and summarizing data. Refactor to avoid assumptions like non-empty input.
	``python
	# utils.py
	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:]]
	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()
	``"

Output:

"To improve the robustness of the code provided, we can add input validation and handle various edge cases such as empty files, invalid numeric values, and out-of-bounds column indices. Below is the refactored version of your code:
 	
``python
# utils.py 	
import csv
 	
def read_csv(filepath):
"""Reads a CSV file and returns its content as a list of rows."""
try:
with open(filepath, 'r') as f:
reader = csv.reader(f)
data = list(reader)
if not data: # Check if the file is empty
raise ValueError(""The CSV file is empty."")
return data
except FileNotFoundError:
raise FileNotFoundError(f""The file '{filepath}' was not found."")
except Exception as e:
raise Exception(f""An error occurred while reading the file: {str(e)}"")
 	
def summarize_column(data, index):
"""Summarizes the specified column in the data."""
if index < 0 or index >= len(data[0]):
raise IndexError(""Column index is out of bounds."")
 	
# Extract the column values, skipping the header 	
values = []
for row in data[1:]:
try:
values.append(float(row[index]))
except ValueError:
raise ValueError(f""Invalid value found in column {index}: '{row[index]}'"")
 	
if not values: # Check if there are no valid values
raise ValueError(""No valid numeric values found in the specified column."")
 	
total = sum(values)
	"To ensure robust parsing and summarization of the input data, we should handle potential issues such as non-existent files, empty files, or missing columns. Here's a refactored version of your code with added checks and error handling:
	
	``python
	# utils.py
	import csv
	
	def read_csv(filepath):
	try:
	with open(filepath, 'r') as f:
	reader = csv.reader(f)
	return list(reader)
	except FileNotFoundError:
	print(f""Error: The file {filepath} was not found."")
	return []
	except Exception as e:
	print(f""Error reading file {filepath}: {e}"")
	return []
	
	def summarize_column(data, index):
	if not data:
	print(""Error: No data to summarize."")
	return None, None
	
	if index < 0 or index >= len(data[0]):
	print(f""Error: Invalid column index {index}."")
	return None, None
	
	try:
	values = [float(row[index]) for row in data[1:]]
	except ValueError:
	print(f""Error: Non-numeric data found in column {index}."")
	return None, None
	
	if not values:
	print(""Error: No data to summarize."")
	return None, None
	
	total = sum(values)
	avg = total / len(values)
	return total, avg
	
	def main():

avg = total / len(values)
return total, avg
 	
def main():
filepath = 'data.csv'
try:
data = read_csv(filepath)
total, avg = summarize_column(data, 1) # Assuming we're interested in the second column (index 1)
print(""Total:"", total)
print(""Average:"", avg)
except Exception as e:
print(""Error:"", str(e))
 	
if __name__ == '__main__':
main()
``
 	
### Changes Made: 	
1. **Error Handling in `read_csv`:** Added error handling to manage file not found and empty file scenarios.
2. **Column Index Validation:** Added checks in `summarize_column` to ensure the column index is within bounds.
3. **Value Validation:** Added validation for each value in the specified column to ensure they can be converted to float. If a value can't be converted, it raises a ValueError with a descriptive message.
4. **Empty Values Handling:** Added a check to see if any valid numeric values were collected from the column, raising an error if none exist.
5. **General Exception Handling:** Wrapped the main logic in a try-except block to catch and print any exceptions that occur.
 	
This refactored version provides clearer feedback when something goes wrong and avoids making assumptions about the input data."
	filepath = 'data.csv'
	data = read_csv(filepath)
	
	if not data:
	print(""No data available for summarization."")
	return
	
	total, avg = summarize_column(data, 1)
	
	if total is not None and avg is not None:
	print(""Total:"", total)
	print(""Average:"", avg)
	
	if __name__ == '__main__':
	main()
	``
	
	### Key Changes:
	1. **File Reading**:
	- Added error handling for `FileNotFoundError` and other potential exceptions.
	- Return an empty list if an error occurs to ensure subsequent logic doesn't fail.
	
	2. **Column Summarization**:
	- Checked if `data` is empty.
	- Checked if the provided column index is valid.
	- Added a try-except block to handle non-numeric data in the specified column.
	- Checked if `values` is empty after filtering out the header row.
	
	3. **Main Function**:
	- Added checks to ensure there is data to summarize before attempting to summarize.
	- Only print results if `total` and `avg` are valid.
	
	This refactored code ensures that it handles various edge cases gracefully, making it more robust and less prone to unexpected failures."

Comment	GPT uses structured validation with explicit exceptions, giving clear user feedback and ensuring robust handling of various edge cases. Codestral prioritizes graceful degradation, using print-based error reporting and fallback behaviors to avoid crashes but potentially missing informative errors.

Model Analysis and Comparison: GPT-4o-mini tends to produce more structured and informative exception-based outputs, which enhance clarity and maintainability for developers. In contrast, Codestral-2501 leans toward pragmatic solutions that favor resilience and efficiency, particularly by minimizing failure through graceful degradation and generator usage. Overall, GPT-4o-mini excels in strict robustness and user guidance, while Codestral-2501 prioritizes performance and fault tolerance through simpler control flows.

Task 22 - Code Completion and Robustness Enhancement (Python)

Strategy:	zero-shot

Model:	gpt-4o-mini	Codestral-2501

Prompt:	"Complete the missing cleaning logic and make sure word frequencies are calculated correctly.
	``python
	# file_processor.py
	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()
	``"f	

Output:

To complete the missing cleaning logic in the `clean_line` function, we'll implement code that removes punctuation and converts the text to lowercase. This will ensure that word frequencies are calculated correctly, without being affected by case or punctuation.
 	
Here's the updated code for `file_processor.py`:
 	
``python
# file_processor.py 	
import string
 	
def load_file(filepath):
with open(filepath, 'r') as f:
return f.readlines()
 	
def clean_line(line):
# Remove punctuation and make lowercase 	
line = line.translate(str.maketrans('', '', string.punctuation)) # Remove punctuation
return line.lower().strip() # Convert to lowercase and strip
	To complete the missing cleaning logic and ensure word frequencies are calculated correctly, you need to implement the `clean_line` function to remove punctuation and convert the text to lowercase. Here is the completed code:
	
	``python
	# file_processor.py
	import string
	
	def load_file(filepath):
	with open(filepath, 'r') as f:
	return f.readlines()
	
	def clean_line(line):
	# Remove punctuation and make lowercase
	translator = str.maketrans('', '', string.punctuation)
	cleaned_line = line.translate(translator).lower()
	return cleaned_line
	
	def count_words(lines):

whitespace
 	
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()
``
 	
### Key Changes Made: 	
1. **Punctuation Removal**: Used `str.translate()` combined with `str.maketrans()` to remove all punctuation from the line. The `string.punctuation` contains all punctuation characters, which are removed from the text.
 	
2. **Lowercase Conversion**: Converted the cleaned line to lowercase using `line.lower()` to ensure that words are counted regardless of their case.
 	
3. **Whitespace Stripping**: Used `strip()` to remove any leading or trailing whitespace from the line after cleaning.
 	
This implementation will accurately count word frequencies from the input text in `input.txt`, taking care of punctuation and case sensitivity.
	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()
	``
	
	### Explanation:
	
	1. **`load_file(filepath)`:** This function reads all lines from the specified file and returns them as a list.
	
	2. **`clean_line(line)`:**
	- `str.maketrans('', '', string.punctuation)` creates a translation table that maps each punctuation character to `None`.
	- `line.translate(translator)` removes all punctuation characters from the line.
	- `lower()` converts the line to lowercase.
	
	3. **`count_words(lines)`:** This function processes each line to count word frequencies:
	- `clean_line(line)` cleans the line.
	- `clean.split()` splits the cleaned line into words.
	- For each word, it updates the word count in the `word_counts` dictionary.
	
	4. **`main()`:** This function drives the entire process:
	- Loads lines from `input.txt`.
	- Counts word frequencies.
	- Prints the word frequencies sorted by word.
	
	This implementation ensures that punctuation is removed, and all words are converted to lowercase before counting their frequencies.

Comment	The GPT output correctly implemented the cleaning logic but used a simple direct approach. The Codestral output also implemented the logic well, though both solutions were nearly identical and lacked additional robustness measures like handling numbers or empty lines.

Strategy:	prompt chaining	
Model:	gpt-4o-mini	Codestral-2501
Prompt	"Step 1: Describe how to normalize and clean a line of text.	

	Step 2: Integrate the cleaned lines into the word counting logic.
	``python
	# file_processor.py
	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()
	``"

Output:

To normalize and clean a line of text, you typically want to perform the following steps:
 	
1. **Remove Punctuation**: Strip out any punctuation marks from the text, as they don't contribute to the meaning of the words for counting purposes.
2. **Convert to Lowercase**: Transform all characters to lowercase to ensure that words are counted in a case-insensitive manner.
3. **Trim Whitespace**: Remove any leading or trailing whitespace to avoid counting empty strings.
 	
Now, let's implement the `clean_line` function to incorporate these steps, and then integrate it into the `count_words` function in your provided code.
 	
Here's the updated code:
 	
``python
# file_processor.py 	
import string
 	
def load_file(filepath):
	"To normalize and clean a line of text, we typically want to remove punctuation, convert the text to lowercase, and possibly handle other common text cleaning steps like removing extra whitespace. We can then integrate this cleaning process into the word counting logic.
	
	Here's how you can implement the `clean_line` function and integrate it into the provided code:
	
	### Step 1: Describe how to normalize and clean a line of text
	
	1. **Remove Punctuation**: Use Python's `string.punctuation` to identify and remove punctuation marks from the text.
	2. **Convert to Lowercase**: Use the `lower()` method to convert all characters in the string to lowercase.
	3. **Split the Line into Words**: Use the `split()` method to divide the cleaned line into individual words.
	4. **Count Words**: Use a dictionary to count the occurrences of each word.
	
	### Step 2: Integrate the cleaned lines into the word counting logic
	
	Here is the complete code with the `clean_line` function implemented

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

def clean_line(line):
# Remove punctuation and make lowercase
# Translate punctuation to None
line = line.translate(str.maketrans('', '', string.punctuation))
# Convert to lowercase
line = line.lower()
# Trim whitespace
return line.strip()

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()
```

### Explanation of the Changes:

- **`clean_line` Function**:
- The `line.translate(str.maketrans('', '', string.punctuation))` part removes all punctuation from the line.
- The `line.lower()` method converts the entire line to lowercase.
- Finally, `line.strip()` removes any leading or trailing whitespace.

This approach ensures that the text is normalized for accurate word counting, allowing you to handle variations in punctuation and casing effectively. The `count_words` function then uses this cleaned line to split into words and count their occurrences.
````

````
and integrated:

```python
# file_processor.py
import string

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

def clean_line(line):
    # Remove punctuation by translating punctuation characters to None
    translator = str.maketrans('', '', string.punctuation)
    cleaned_line = line.translate(translator)
    # Convert to lowercase
    cleaned_line = cleaned_line.lower()
    return cleaned_line

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

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

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

### Explanation

1. **`clean_line` Function**:
    - `str.maketrans('', '', string.punctuation)` creates a translation table that maps each punctuation character to `None`.
    - `line.translate(translator)` removes all punctuation from the line.
    - `line.lower()` converts the entire line to lowercase.

2. **Integration into `count_words`**:
    - The function iterates over each line, cleans it using `clean_line`, and then splits the cleaned line into words.
    - It updates the word count dictionary for each word encountered.

### Usage
- Ensure you have an `input.txt` file in the same directory as the script, or modify the `filepath` variable to point to the correct file location.
````
````

	- When you run the script, it will load the file, clean and count the words, and print the results sorted alphabetically. This approach ensures that the text is normalized and cleaned before counting, making the word count logic more accurate and reliable."
Comment	The GPT output demonstrated clearer reasoning through a step-by-step breakdown before implementation, showing stronger contextual understanding. Codestral followed the same logic but lacked the structured explanation, making GPT's output slightly more informative.

Model Analysis and Comparison: GPT-4o-mini consistently provided more structured explanations and demonstrated better reasoning in both the zero-shot and prompt chaining strategies. While Codestral-2501 produced functionally correct code in both cases, its outputs were less detailed and missed the stepwise breakdown that GPT offered in the chaining strategy. Overall, GPT showed stronger capability in integrating prompt structure into coherent, readable, and robust code completions.