1. **Code Summarization (Java)**

Goal: Summarize the functionality of the following method

Zero-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

Summarize the functionality of the following method:

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

Output	**Summary of method functionality:** The countWordFrequency method takes a list of strings (words) and returns a map (Map<String, Integer>) where each unique word from the list is a key, and its value is the number of times that word appears in the list. In other words, it counts the frequency of each word in the input list and returns the results as a mapping from word to frequency.	The method countWordFrequency takes a list of strings (List<String>) as input and returns a Map<String, Integer> that represents the frequency of each word in the input list. Here's a breakdown of its functionality: 1. **Initialization**: ○ A new HashMap named freqMap is created to store the word frequency. The keys are words (strings), and the values are their corresponding frequencies (integers). 2. **Iteration**: ○ The method iterates through each word in the input list words. ○ For each word, it updates the freqMap: ■ If the word is already present in the map, its value (frequency) is incremented by 1. ■ If the word is not present in the map, its value is set to 1 (default value). 3. **Return**: ○ After processing all words in the list, the method returns the freqMap containing the frequency of each word. In summary, this method counts how many times each word appears in the input list and stores these counts in

		a map, which is then returned.
Comment	The output summarizes the method well and covers all the main points of the method, without going into too much detail.	The output not only summarizes the method briefly, but delves into the method and explains in detail each part of the method.

Chain-of-Thought

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	
Prompt	Let's break down the method step-by-step. First, it initializes a frequency map. Then, it iterates through the list of words. For each word, it updates the frequency map by checking if the word is already present. After the loop, it returns the frequency map. Now, summarize the functionality of the following method in a concise manner.	

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

Output	The countWordFrequency method takes a list of words and returns a map containing each unique word as a key and the number of times it appears in the list as the value.	The method countWordFrequency takes a list of words and returns a frequency map (a Map<String, Integer>) that counts the occurrences of each word in the list. It initializes an empty frequency map, iterates through each word in the list, and updates the map by increasing the count of each word, using getOrDefault to handle the case where the word might not already be in the map. After processing all words, it returns the populated frequency map.
Comment	This output summarizes the code more briefly than the zero-shot prompting, following the instructions by making it "concise."	This output also summarizes the code more briefly than the zero-shot prompting, shortening the step-by-step analysis to sentences.

Model Comparison and Analysis

GPT-4.1 produced more short and concise outputs than Codestral 25.01 in both the zero-shot and chain-of-thought prompting strategies. However, while GPT-4.1 made its output concise, Codestral 25.01 included details to ensure the code was easily understandable. Chain-of-thought prompting shortened the outputs of both models compared to zero-shot prompting at the cost of losing detail. Depending on the user's needs, both prompting strategies can be good, and neither are objectively better than each other.

2. **Bug Fixing (Python - Off-by-One)**

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

Few-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

Example (Python)
Buggy code:

```
def inclusive_sum(a,b):
	total=0
	for i in range(a,b):
		total+=i
	return total
```

Fixed explanation:
Use range(a,b+1) or range(a,b) with b incremented.

Your task: identify and fix the off-by-one error in the next function, then show the corrected code:

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

Output

gpt-4.1

Buggy code:

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

Fixed explanation:
The range(start, end) function does **not include** end in the loop. To make the summation **inclusive** of both start and end, use range(start, end + 1).

Corrected code:

Codestral-2501

The off-by-one error in the given function sum_range occurs because the range function in Python is exclusive of the end value. This means that the loop will iterate from start up to, but not including, end.

To fix this, you need to adjust the range so that it includes the end value. You can do this by either using range(start, end + 1) or by incrementing end when calling the range function.

Here is the corrected code:

Using range(start, end + 1)

```
def sum_range(start, end):
    total = 0
```

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

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

Incrementing end

```
def sum_range(start, end):
    total = 0
    for i in range(start, end):
        total += i
    return total
# Example usage
result = sum_range(1, 5)  # This will now include 5 in the sum
print(result)  # Output should be 15 (1+2+3+4+5)
```

Both approaches will correctly sum the range from start to end inclusive. The first approach is more idiomatic in Python, while the second approach modifies the loop logic slightly.

Comment	The output cleanly pinpoints the off-by-one issue, offers a succinct explanation, and presents a minimal, correct code snippet—very clear and to the point.	The few-shot solution reliably detects primes but does so inefficiently by testing every number up to n−1; the model succinctly acknowledges correctness while clearly flagging the optimization opportunity.

Zero-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

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

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

<table>
<tr>
<td>

Output

</td>
<td>

Let's analyze the function:

```
def sum_range(start, end):

    total = 0

    for i in range(start, end):

        total += i

    return total
```

What does it currently do?

- It sums all numbers from start **up to but not including** end.

What's the likely intent?

- Often, "sum from start to end" means **inclusive** of both endpoints.

Off-by-one error:

- Python's range(start, end) is **exclusive** of end.
- So if you want end **included**, you should write range(start, end+1).

Fixed function:

```
def sum_range(start, end):

    total = 0

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

        total += i

    return total
```

Example:

- sum_range(1, 5)
    - **Original:** Sums 1+2+3+4 = **10**
    - **Fixed:** Sums 1+2+3+4+5 = **15**

</td>
<td>

The off-by-one error in the function sum_range occurs because the range(start, end) function generates a sequence of numbers starting from start and ending just before end. This means that the last number included in the sequence is end - 1, which does not contribute to the sum if end is the intended upper limit.

To fix this off-by-one error, you should modify the range to include the end value. Here is the corrected function:

```
def sum_range(start, end):

    total = 0

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

        total += i

    return total
```

By changing range(start, end) to range(start, end + 1), the loop now includes the end value in the sum.

Here is an example to illustrate the difference:

Original Function

```
def sum_range(start, end):

    total = 0

    for i in range(start, end):

        total += i

    return total



print(sum_range(1, 5))  # Outputs: 10 (1 + 2 + 3 + 4)
```

Corrected Function

```
def sum_range(start, end):

    total = 0

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

        total += i

    return total



print(sum_range(1, 5))  # Outputs: 15 (1 + 2 + 3 + 4 + 5)
```

With the corrected function, the output will be 15, which

</td>
</tr>
</table>

		is the correct sum of the numbers from 1 to 5.
Comment	The output methodically breaks down the function’s behavior, intent, and error, then demonstrates the fix with an example—very clear and thorough, albeit a bit verbose for a simple off-by-one correction.	The output accurately diagnoses the off-by-one issue, delivers a correct fix, and uses clear examples to illustrate the change—very thorough. It could be more concise by avoiding the repeated original-versus-fixed code blocks.

Model Comparison and Analysis

GPT-4.1’s few-shot explanation is crisp and on-point, while its zero-shot version trades brevity for a thorough behavioral breakdown with examples. In contrast, Codestral 25.01 misaligned on the few-shot (discussing primes instead of the sum_range bug) but its zero-shot correctly diagnoses and illustrates the fix, albeit with some redundant repetition.

3. Bug Classification (C++)

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

Chain-of-Thought

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	
Prompt	Let's walk through this function. The function declares a local array arr and returns a pointer to it. However, returning a pointer to a local variable is risky because the array's memory will be deallocated once the function exits. Identify the type of bug here.	

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

Output	The bug in the function you've shown is known as a **dangling pointer** (or using a pointer to a local variable that has gone out of scope). **Explanation** When you declare `int arr[size];` inside the function, `arr` is a **local (automatic) variable**. Its memory is allocated **on the stack** and is only valid **inside the function's scope**. When the function returns,	The bug in the function getArray is related to the lifetime of the local variable arr and the pointer returned by the function. Let's break it down: 1. **Local Array Declaration**: The array arr is declared within the function getArray. This means arr is a local variable and is allocated on the stack. 2. **Returning a Pointer**: The function returns a pointer to the local array arr.

the stack space used by `arr` is reclaimed, and the pointer you return (`arr`) now points to an invalid (dangling) memory location.

Type of Bug

Dangling pointer (or **returning address of local variable**):
Returning a pointer to a local variable causes **undefined behavior** when you try to use that pointer outside the function.

How to Fix

1. *(Remember to `free()` the memory when done.)*
2. **Pass an array to be filled as an argument** instead of creating it inside the function.

Summary Table:

Bug Type	Description
Dangling pointer	Returning address of local (stack) variable
Undefined behavior	Using returned pointer outside the function is unsafe

---

In short:
You are returning a **dangling pointer** (pointer to deallocated stack memory).

Use dynamic memory allocation (heap):

```
int* getArray(int size) {

   int* arr = malloc(size * sizeof(int));

   return arr;

}
```

3. **Lifetime of Local Variables**: When the function getArray exits, the stack frame associated with the function is destroyed. This means the memory allocated for arr is deallocated, and any pointer pointing to that memory becomes invalid.
4. **Dangling Pointer**: Returning a pointer to a local variable that is no longer valid after the function exits results in a dangling pointer. Using this pointer in any way (e.g., dereferencing it) will lead to undefined behavior.

Type of Bug

This is a **dangling pointer bug**. Specifically, it falls under the category of **use-after-free** or **use-after-scope** because the pointer is being used after the variable it points to has gone out of scope.

Correct Approach

To avoid this bug, you should dynamically allocate memory for the array using malloc or new (in C++) and ensure that the caller of the function is responsible for freeing the allocated memory. Here is an example of how you can do this in C:

```
#include <stdlib.h>



int* getArray(int size) {

    int* arr = (int*)malloc(size * sizeof(int)); // Dynamically allocate memory

    if (arr == NULL) {

        // Handle memory allocation failure

        return NULL;

    }

    return arr; // Return the pointer to the dynamically allocated array

}
```

Comment	The chain-of-thought walkthrough is very thorough—clearly tracing stack allocation, naming the bug as a dangling pointer, and offering both a summary table and a dynamic-allocation fix. It's highly educational, though the repeated recaps (table plus "In short" section) could be trimmed for conciseness.	The chain-of-thought clearly unpacks each step—stack allocation, pointer return, scope exit—and accurately labels it a dangling-pointer (use-after-scope) bug. It then offers a complete heap-allocation remedy with example code. Very instructive, though the detailed breakdown repeats similar points and could be tightened for brevity.

Few-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

Example (C++)
Buggy function:

```
int* bad(){ int x=5; return &x; }
```

Bug classification:
Returning pointer to local (stack) variable ➜ **dangling pointer / use-after-scope**.

Given the next C++ function, classify the main bug type in one phrase:

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

Output	**Returning pointer to local (stack) variable ➜ dangling pointer / use-after-scope**	The main bug type in the given C++ function is: **Returning pointer to local (stack) variable ➜ dangling pointer / use-after-scope**.
Comment	The few-shot output perfectly mirrors the example's style, giving a concise, accurate one-phrase classification—clear, consistent, and exactly on target.	The output sticks precisely to the few-shot template—succinctly labels the bug in one phrase, mirrors the example's style, and conveys the classification with perfect brevity and clarity.

Model Comparison and Analysis

Both models excel under chain-of-thought prompting by delivering deep, step-by-step explanations, though each introduces minor redundancy. In few-shot mode, they both shine at producing terse, template-driven answers. It would seem that chain-of-thought is better for comprehensive educational detail and few-shot for rapid, concise classification.

4. Generating Email Validators (Python + Regex)

Goal: Complete the function using regex to validate basic email addresses

Zero-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	

Prompt

Complete the function using regex to validate basic email addresses:

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

Output

Here's a simple implementation using regex to validate basic email addresses:

```
import re



def is_valid_email(email):

    # Basic regex for email validation

    pattern = r'^[\w\.-]+@[\w\.-]+\.\w+$'

    return re.match(pattern, email) is not None
```

Explanation:

- `^` and `$` assert start and end of the string.
- `[\w\.-]+` matches one or more word characters (`a-zA-Z0-9_`), dots, or hyphens for the username part.
- `@` matches the literal `@` symbol.
- `[\w\.-]+` matches the domain name part similarly.
- `\.` matches the dot before the TLD.
- `\w+` matches the TLD (e.g., com, net, org).

This covers many common email formats but is not fully RFC-compliant. Let me know if you want a stricter or

Certainly! Here's a simple implementation of the `is_valid_email` function using regex to validate basic email addresses:

```
import re

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

Explanation:

- `^[a-zA-Z0-9._%+-]+` : Starts with one or more allowed characters for the local part.
- `@` : The "at" symbol separating local and domain parts.
- `[a-zA-Z0-9.-]+` : Domain name with allowed characters.
- `\.` : Literal dot before the TLD.
- `[a-zA-Z]{2,}$` : Top-level domain with at least two alphabetic characters.

Usage example:

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

	more comprehensive regex!	This pattern covers most common email formats but keep in mind that email validation can get very complex if you aim for complete RFC compliance. For many practical purposes, this suffices.
Comment	The output delivers a clear, concise regex-based email validator with well-commented code and a helpful breakdown of each pattern component. It appropriately notes its limitations (not fully RFC-compliant) while providing a solid solution for basic use.	The output offers a more robust regex with expanded allowed characters, clear inline explanations, and practical usage examples—striking a great balance between readability and functionality—while appropriately noting that full RFC compliance would require even more complexity.

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Repetitions	3	

Prompt

Design a regex email validator. List key parts (username, @, domain, TLD).
Provide candidate pattern and a final function definition.

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

Output

(votes 1/3)

```
import re


def is_valid_email(email):

        """

        Validates an email address using regex.



        Email format:

        - Username: alphanumeric characters, dots, underscores, percent, plus, hyphen

        but no consecutive dots or dots at start/end.

        - Exactly one '@' symbol.

        - Domain: labels separated by dots, labels
```

(votes 1/3) Certainly! To design a regex email validator, we can break down the email address into key parts:

1. **Username**: Allows alphanumeric characters, dots, underscores, and hyphens.
2. **@ symbol**: Separates username and domain.
3. **Domain**: Usually consists of alphanumeric characters and hyphens, separated by dots.
4. **Top-Level Domain (TLD)**: Typically alphabetic, 2-6 characters (e.g., .com, .org, .co.uk).

Candidate Pattern:

A simplified regex pattern could be:

^[A-Za-z0-9._-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}$

Explanation:

- `^`: Start of string
- `[A-Za-z0-9._-]+`: Username with allowed characters
- `@`: Literal at symbol
- `[A-Za-z0-9.-]+`: Domain name with allowed

```
alphanumeric with optional internal hyphens,

        no leading/trailing hyphens.

        - TLD: 2 to 63 letters.


        Returns True if valid, False otherwise.

        """

        pattern = re.compile(

        r'^[a-zA-Z0-9]+([._%+-][a-zA-Z0-9]+)*@'

        r'([a-zA-Z0-9]([a-zA-Z0-9-]*[a-zA-Z0-9])?\.)+'

        r'[a-zA-Z]{2,63}$'

        )

        return bool(pattern.match(email))
```

characters
- `\.`: Dot before TLD
- `[A-Za-z]{2,6}`: TLD with 2 to 6 alphabetic characters
- `$`: End of string

Final Python Function:

```
import re

def is_valid_email(email):
        pattern =
r'^[A-Za-z0-9._-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}$'
        return re.match(pattern, email) is not None
```

Final Answer:

```
def is_valid_email(email):
        pattern =
r'^[A-Za-z0-9._-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}$'
        return re.match(pattern, email) is not None
```

Comment	The output includes a clear docstring detailing username, `@`, domain, and TLD constraints, and implements a robust regex that prevents consecutive or edge dots, enforces hyphen placement, and caps TLD length—making it both thorough and practical (though moving the compiled pattern to module scope could optimize repeated calls).	The output clearly breaks the email into username, `@`, domain, and TLD parts, provides a straightforward candidate regex, and wraps it in a simple function—ideal for basic validation, though it doesn't guard against consecutive dots or more complex domain rules.

Model Comparison and Analysis

GPT-4.1-mini consistently delivers richer, more robust validators—complete with detailed docstrings, edge-case handling (e.g. consecutive-dot prevention, hyphen placement), and optimization notes—whereas GPT-4.1-nano favors concise, straightforward patterns and explanations that cover the common cases but omit deeper nuance.

5. Generating Flask APIs (Python)

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

Few-Shot

Model	gpt-4.1-mini	gpt-4.1-nano

Temperature	0.7
Token Limit	1024

Prompt

Example (Flask)

```
@app.route('/hello/<name>')
def hello(name):
        return jsonify({'msg':f'Hello {name}!'})
```

Following the pattern, create a '/greet/<username>' endpoint that returns JSON {"greeting":"Hello, <username>!"}:

```
from flask import Flask, jsonify
app = Flask(__name__)

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

Output

```
from flask import Flask, jsonify

app = Flask(__name__)



@app.route('/greet/<username>')

def greet(username):

        return jsonify({'greeting': f'Hello, {username}!'})
```

Certainly! You want to create an endpoint `/greet/<username>` that responds with JSON containing a greeting message. Following the pattern you provided, here's the implementation:

```
from flask import Flask, jsonify
app = Flask(__name__)

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

This function captures the `<username>` from the URL and returns a JSON object with the key `"greeting"` and the value `"Hello, <username>!"`.

Comment	The implementation precisely follows the example, defining the `/greet/<username>` route and returning the correct JSON with `jsonify` and an f-string. It's concise, accurate, and idiomatic Flask code.	The response accurately implements the `/greet/<username>` route and JSON output, and the added explanatory preamble clearly restates the task. It's functionally identical to the minimal version—with slightly more prose—but remains clear, correct, and idiomatic Flask code.

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Repetitions	3	

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

Output	(votes 2/3)
	
	from flask import Flask, jsonify
	
	app = Flask(__name__)
	
	@app.route('/greet/<username>')
	
	def greet(username):
	
	    return jsonify({"message": f"Hello, {username}!"})
		(votes 2/3)
		
		from flask import Flask, jsonify
		app = Flask(__name__)
		
		@app.route('/greet/<username>')
		def greet(username):
		    return jsonify({"message": f"Hello, {username}!"})

Comment	The code correctly defines the `/greet/<username>` route and uses `jsonify` to return a personalized message, but it uses the key `"message"` instead of the expected `"greeting"` and doesn't include the reasoning about route declaration, parameter capture, or the `jsonify` call that the prompt requested.	The implementation correctly sets up the route and `jsonify` call, but it uses the key `"message"` instead of the specified `"greeting"` and omits any reasoning about the route, parameter capture, or use of `jsonify` that the prompt requested.

Model Comparison and Analysis

Nano's few-shot email validator is a bit more robust with examples, while mini's is clear but simpler; in self-consistency mode, mini adds edge-case handling and optimization tips, whereas nano stays basic. Both models nail the Flask route in few-shot but miss the self-consistency spec by using the wrong JSON key and skipping the requested reasoning.

6. SQL Schema Design (SQL)

Goal: Write the schema for a review app with users, books, and reviews:

Chain-of-Thought

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Let's start by defining the tables needed for this app. We need a table for users, with fields like id and name, and a table for books, with fields like id and title. Lastly, we need a table for reviews, which links users and books with a rating. Now, write the schema for this. -- TODO: Design schema with appropriate keys and constraints -- Tables: users(id, name), books(id, title), reviews(id, user_id, book_id, rating)	

Output

gpt-4.1-mini:

```
CREATE TABLE users (

    id SERIAL PRIMARY KEY,

    name VARCHAR(255) NOT NULL

);



CREATE TABLE books (

    id SERIAL PRIMARY KEY,

    title VARCHAR(255) NOT NULL

);



CREATE TABLE reviews (

    id SERIAL PRIMARY KEY,

    user_id INTEGER NOT NULL,

    book_id INTEGER NOT NULL,

    rating INTEGER NOT NULL CHECK (rating >= 1 AND rating <= 5),

    CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,

    CONSTRAINT fk_book FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE,

    CONSTRAINT unique_user_book_review UNIQUE (user_id, book_id)
```

gpt-4.1-nano:

Certainly! Here's a well-structured schema for your app with appropriate keys and constraints:

```
-- Users table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);

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

-- Reviews table
CREATE TABLE reviews (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    book_id INT NOT NULL,
    rating INT CHECK (rating >= 1 AND rating <= 5),
    -- Foreign key constraints
    CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
    CONSTRAINT fk_book FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE,
    -- Ensure a user reviews a book only once
    UNIQUE (user_id, book_id)
);
```

Explanation:

- `SERIAL` is used for auto-incrementing primary keys.
- `NOT NULL` constraints ensure essential data is always provided.
- `CHECK` constraint on `rating` enforces valid rating values (1-5).
- **Foreign keys** link reviews to users and books, with cascading deletes to maintain referential

	); This schema defines: - `users` with unique `id` and a `name`. - `books` with unique `id` and a `title`. - `reviews` with unique `id`, foreign keys `user_id` and `book_id` referencing `users` and `books` respectively, a `rating` constrained between 1 and 5, and a unique constraint to ensure a user can only review a specific book once. Let me know if you want me to add timestamps or other fields!	integrity. - **Unique constraint** on `(user_id, book_id)` prevents duplicate reviews from the same user for the same book. Let me know if you'd like additional features or modifications!
Comment	The output delivers a complete, normalized schema that meets all requirements—defining primary keys, foreign keys with cascading deletes, rating constraints, and a unique review-per-user-book rule. It's clear and production-ready (you could optionally add timestamps or indexes), making it an excellent solution.	The schema is cleanly organized with auto-incrementing keys, NOT NULL and CHECK constraints, cascading foreign keys, and a unique user-book review rule. The accompanying explanations clarify each choice, making it both comprehensive and ready for production.

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Repetitions	3	
Prompt	Design three SQL tables (users, books, reviews) with PK/FK constraints. Explain choices, then write the DDL. -- TODO : Design schema with appropriate keys and constraints -- Tables : users (id , name) , books (id , title) , reviews (id , user_id , book_id , rating)	
Output	(votes 1/3) To design three SQL tables—**users**, **books**, and **reviews**—with appropriate primary key (PK) and foreign key (FK) constraints, we need to consider the relationships and data integrity: **Explanation of Design Choices** - **users**: Stores user information. Each user has a unique identifier (`user_id`). This table is the source of users who can write reviews. - **books**: Stores book information. Each book has a unique identifier (`book_id`).	(votes 1/3) CREATE TABLE users (user_id INT PRIMARY KEY AUTO_INCREMENT, username VARCHAR(50) NOT NULL UNIQUE, email VARCHAR(100) NOT NULL UNIQUE, created_at TIMESTAMP DEFAULT

- **reviews**: Stores reviews that users write about books. Each review:
    - Has a unique identifier (`review_id`).
    - References a user (`user_id`) who wrote it.
    - References a book (`book_id`) that is being reviewed.
    - May contain a rating and review text.

Constraints:

- Primary keys on each table ensure unique identification.
- Foreign keys on `reviews.user_id` and `reviews.book_id` ensure referential integrity to `users` and `books`.
- Optionally, you might want to prevent a user from reviewing the same book multiple times (a unique constraint on `(user_id, book_id)`).

Final Answer:

```
CREATE TABLE users (

        user_id INT PRIMARY KEY AUTO_INCREMENT,

        username VARCHAR(50) NOT NULL UNIQUE,

        email VARCHAR(100) NOT NULL UNIQUE,

        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);


CREATE TABLE books (

        book_id INT PRIMARY KEY AUTO_INCREMENT,

        title VARCHAR(255) NOT NULL,

        author VARCHAR(100) NOT NULL,

        published_year INT,

        isbn VARCHAR(20) UNIQUE

);
```

```
CURRENT_TIMESTAMP

);


CREATE TABLE books (

        book_id INT PRIMARY KEY AUTO_INCREMENT,

        title VARCHAR(255) NOT NULL,

        author VARCHAR(100) NOT NULL,

        published_year INT

);


CREATE TABLE reviews (

        user_id INT,

        book_id INT,

        review_text TEXT,

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

        review_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

        PRIMARY KEY (user_id, book_id),

        FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,

        FOREIGN KEY (book_id) REFERENCES books(book_id) ON DELETE CASCADE

);
```

Notes:

- `AUTO_INCREMENT` (or equivalent) is used for `user_id` and `book_id` to generate unique identifiers.
- `UNIQUE` constraints on `username` and `email` prevent duplicates.
- The `reviews` table uses a composite primary key (`user_id`, `book_id`) to prevent multiple reviews by the same user for a single book.
- `ON DELETE CASCADE` ensures that when a user or book is deleted, associated reviews are

```
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),

            review_text TEXT,

            review_date TIMESTAMP DEFAULT
CURRENT_TIMESTAMP,

            CONSTRAINT fk_review_user FOREIGN KEY
(user_id) REFERENCES users(user_id) ON DELETE
CASCADE,

            CONSTRAINT fk_review_book FOREIGN KEY
(book_id) REFERENCES books(book_id) ON DELETE
CASCADE,

            CONSTRAINT unique_user_book_review
UNIQUE (user_id, book_id)

);
```

	Notes: • `AUTO_INCREMENT` (MySQL) or equivalent used to generate unique IDs. • `username` and `email` are unique to avoid duplicates. • `isbn` is unique in `books`. • `rating` is constrained to a 1-5 scale. • `ON DELETE CASCADE` on foreign keys means that if a user or book is deleted, their reviews are also deleted automatically. • The unique constraint on `(user_id, book_id)` ensures a user cannot review the same book multiple times.	also removed, maintaining referential integrity. • The `rating` field includes a check constraint to ensure ratings are between 1 and 5.
Comment	Defines normalized tables with PKs, cascading FKs, rating checks, and a unique user–book review constraint, plus extras (timestamps, author, ISBN,	Auto-increment PKs for users/books with unique username/email; reviews uses a composite PK (user_id, book_id), rating check, cascading FKs, and a

	email)—production-ready but beyond the original spec.	timestamp—robust and clear.

Model Comparison and Analysis

In chain-of-thought mode, both GPT-4.1-mini and GPT-4.1-nano produce fully normalized schemas with PKs, FKs, checks, and uniqueness rules—but the mini version goes a step further by suggesting optional timestamps or indexes, while the nano version stays strictly within the original field list. In self-consistency mode, the mini model again enriches the design with extra columns (author, ISBN, email) making it very production-ready but beyond spec, whereas the nano model remains concise and focused, delivering exactly the required tables and constraints.

7. Null Dereference Detection (Java)

Goal: Identify any null dereference risk:

Few-Shot

<table>
<tr><td><b>Model</b></td><td>gpt-4.1</td><td>Codestral-2501</td></tr>
<tr><td><b>Temperature</b></td><td colspan="2">0.7</td></tr>
<tr><td><b>Token Limit</b></td><td colspan="2">1024</td></tr>
<tr><td><b>Prompt</b></td><td colspan="2">Example (Java null-check)
if(s == null) return 0; else return s.length();

Task: point out the null-dereference risk in the following method and suggest a safe fix:
<pre>
public int getLength(String s){
        return s.length();
}
</pre></td></tr>
<tr><td><b>Output</b></td><td><b>Null-dereference risk:</b>
In the method<pre>
public int getLength(String s){

    return s.length();

}
</pre>if the argument <code>s</code> is <code>null</code>, calling <code>s.length()</code> will throw a <code>NullPointerException</code>.<hr><b>Safe fix:</b></td><td>The method <code>getLength(String s)</code> is at risk of a <code>NullPointerException</code> if the input string <code>s</code> is <code>null</code>. This happens because the <code>length()</code> method on a <code>String</code> object is called without first checking if <code>s</code> is <code>null</code>.

To fix this, you need to add a null-check before attempting to call <code>s.length()</code>. Here's how you can do it:<pre>
public int getLength(String s) {

    if (s == null) {
</pre></td></tr>
</table>

	Add a null-check before calling `.length()`:
	public int getLength(String s) {
	if (s == null) return 0;
	else return s.length();
	}
	or more succinctly:
	public int getLength(String s) {
	return (s == null) ? 0 : s.length();
	}
	Summary:
	The original code risks a `NullPointerException` if `s` is null. The fix is to check for `null` before dereferencing.
		return 0;
		}
		return s.length();
		}
		This way, if `s` is `null`, the method will return `0` instead of throwing a `NullPointerException`. If `s` is not `null`, it will return the length of the string as intended. This ensures that the method is safe from null-dereference risks.

Comment	The response clearly identifies the NullPointerException risk, provides both a verbose and a ternary-operator fix, and summarizes succinctly—very comprehensive, though the two fix options slightly duplicate effort.	This version concisely explains the risk and presents a single, clean null-check solution—straightforward and effective, but missing the alternate terse form.

Prompt-Chaining

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024 (512 per response)	

Prompt 1	Explain what can go wrong if s is null in the Java method.
	public int getLength(String s){
	return s.length();
	}	

Output 1	If the parameter `s` is `null` when the method `getLength` is called, the following problem will occur: • The method tries to call `s.length()`. • Since `s` is `null`, there is no actual `String` object to invoke `length()` on. • This causes Java to throw a	If the parameter `s` is `null` when calling the `getLength` method, a `NullPointerException` will be thrown at runtime. This is because the method attempts to invoke `s.length()`, which is an instance method on the `String` object. If `s` is `null`, there is no actual `String` object to invoke the method on, leading to a null reference dereference.

`NullPointerException` at runtime.

In other words, the method does not handle the `null` case, leading to a runtime crash or exception when `null` is passed in.

How to fix this issue?

You can add a null check before calling `length()`:

```
public int getLength(String s){

   if (s == null) {

      return 0; // or handle null case as appropriate

   }

   return s.length();

}
```

Or use Java 8's `Optional` or other patterns to safely handle null inputs.

What can go wrong:

- **NullPointerException:** The most common issue is that your program will throw a `NullPointerException`, which can cause the program to terminate unexpectedly unless properly handled.
- **Potential crashes or unpredictable behavior:** If not caught, this exception can lead to application crashes or inconsistent states.
- **Difficulty in debugging:** If the method is called with `null` input unexpectedly, it might be challenging to identify the cause, especially if the null value propagates through multiple layers.

How to prevent this:

- Validate input before calling `length()`:

```
public int getLength(String s){

   if (s == null) {

      // Handle null case, e.g., return 0 or throw an exception

      return 0; // or throw new IllegalArgumentException("Input string cannot be null");

   }

   return s.length();

}
```

- Use `Objects.requireNonNull(s)` to enforce non-null input:

```
import java.util.Objects;



public int getLength(String s){

   Objects.requireNonNull(s, "Input string cannot be null");

   return s.length();

}
```

Summary: Passing `null` to `getLength` causes a `NullPointerException` at runtime when calling `s.length()`. Proper null checks or input validation can prevent this issue.

Prompt 2	Using that explanation: {analysis} write a safe implementation that avoids NullPointerException.	

Output 2

Here's a safe implementation of the `getLength` method that avoids a `NullPointerException` by checking if `s` is `null` before calling `length()`:

```
public int getLength(String s) {

    if (s == null) {

        return 0; // or any other appropriate default value

    }

    return s.length();

}
```

This way, if `s` is `null`, the method returns `0` instead of throwing an exception.

Certainly! To make the `getLength` method safe and avoid `NullPointerException`, you should include a null check before calling `s.length()`. Here's a robust implementation:

```
public int getLength(String s) {

    if (s == null) {

        // Decide how to handle null input:

        // For example, return 0 if null input should be treated as empty.

        // Alternatively, throw an exception to indicate invalid input.

        return 0;

        // Or, uncomment the following line to throw an exception:

        // throw new IllegalArgumentException("Input string cannot be null");

    }

    return s.length();

}
```

Explanation:

- Checks if `s` is `null`.
- If `null`, returns `0` (or handles as appropriate).
- If not `null`, safely returns the length of the string.

This approach ensures the method won't throw a `NullPointerException` regardless of the input.

Comment	The output1 clearly explains the NullPointerException risk, steps through the failure scenario, and offers a simple null-check fix plus Optional-based alternatives—very instructive, though a bit wordy. The output2 is a succinct, direct implementation that cleanly addresses the prompt with a null-check guard—ideal for quick reference, albeit without broader	The output1.1 delivers an in-depth analysis of crash scenarios, debugging challenges, and multiple prevention patterns (null-check and `Objects.requireNonNull`), making it highly thorough but somewhat verbose. The output2.1 provides a robust, commented fix with optional exception-throwing, accompanied by a clear explanation—well-balanced between detail and clarity,

	context.	though slightly longer than necessary.

Model Comparison and Analysis

In few-shot mode, GPT-4.1 offers a more exhaustive explanation—identifying both the NullPointerException risk and two fix styles—whereas Codestral-25.01 delivers a tighter, single-solution response but omits the alternate terse form. Under prompt-chaining, GPT-4.1-mini strikes a balance between instructive detail and succinctness—clearly critiquing both verbose and concise fixes—while GPT-4.1-nano goes deeper into edge-case analysis and alternative patterns at the expense of added verbosity.

8. CSV Parser Variants (Python)

Goal: Improve the parser to support quoted fields:

Prompt-Chaining

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024 (512 per response)	
Prompt 1	List corner-cases a CSV line parser must handle when fields are quoted.	

```
def parse_csv_line(line):
        return line.split(',')  # broken
```

Output 1	When parsing CSV lines with quoted fields, a simple split on commas is insufficient because commas inside quoted fields are not separators. Here are key corner cases a CSV line parser must handle when fields are quoted: 1. **Commas inside quoted fields** ○ Example: `"Smith, John",25,"New York"` The comma inside `"Smith, John"` should not split the field. 2. **Quotes inside quoted fields (escaped quotes)** ○ Example: `"John ""Johnny"" Smith",30,"NY"` Double quotes `""` inside a quoted field represent a literal quote character.	When parsing CSV lines with quoted fields, a robust parser must handle several corner cases to correctly interpret the data. Here are the key corner cases: 1. **Quoted Fields Containing Commas** ○ Fields that include commas inside quotes should not be split at those commas. ○ Example: `"Smith, John",25,"New York, NY"` 2. **Quoted Fields Containing Double Quotes** ○ Double quotes inside a quoted field are escaped by doubling them (`""`). ○ Example: `"John ""Johnny"" Smith",30` 3. **Unclosed Quotes**

	3. **Fields that are fully quoted or partially quoted** ◦ Fields may be fully enclosed in quotes or not quoted at all. 4. **Empty fields** ◦ Example: `"John",,"NY"` Empty fields between commas. 5. **Whitespace outside quotes** ◦ Whitespace before or after fields that may or may not be significant. 6. **Newlines inside quoted fields** ◦ Example: `"John\nSmith",30,"NY"` Quoted fields may contain newlines if the CSV spans multiple lines. 7. **Fields with only quotes** ◦ Example: `"""",25,"NY"` A field that is an empty quoted string or contains only quotes. 8. **Trailing commas (empty last field)** ◦ Example: `"John",25,"NY",` The trailing comma indicates a final empty field. 9. **Malformed lines** ◦ Unclosed quotes, mismatched quotes, or other syntax errors. --- A robust CSV parser must handle these cases correctly, not just split on commas. --- If you want, I can provide a corrected `parse_csv_line` function that handles quoted fields properly.	◦ Lines where a quoted field is not properly closed. ◦ Example: `"John,25,"New York"` (missing closing quote) 4. **Empty Fields** ◦ Fields that are empty, either quoted or unquoted. ◦ Example: `John,,25` 5. **Fields with Leading or Trailing Spaces** ◦ Spaces outside quotes should be handled correctly, either preserved or trimmed based on requirements. ◦ Example: `John , " New York "` 6. **Fields with Whitespace Inside Quotes** ◦ Spaces inside quotes are part of the data. ◦ Example: `" John Doe "` 7. **Line Ending Variations** ◦ Handling different line endings (`\n`, `\r\n`, `\r`) if parsing multiple lines. 8. **Escape Sequences and Special Characters** ◦ Handling escape sequences if the CSV format uses them (less common in standard CSV). 9. **Multiple Consecutive Delimiters** ◦ Consecutive commas indicating empty fields. ◦ Example: `John,,New York` 10. **Fields with Embedded Line Breaks** ◦ Quoted fields that span multiple lines (if parsing multi-line CSV data). --- **Summary:** A proper CSV parser must recognize when a comma is inside quotes versus when it separates fields, handle escaped quotes inside quoted fields, detect unclosed quotes, and accurately process empty or whitespace-only fields, among other edge cases. The naive split on commas (`line.split(',')`) fails to address these complexities, leading to incorrect parsing.
Prompt 2	Refactor the function below using the corner-cases list: {analysis}	

	so it returns an accurate list of fields.

Output 2

Here's a refactored `parse_csv_line` function that handles all the corner cases you listed. It parses the CSV line character-by-character, respects quoted fields, handles escaped quotes, empty fields, whitespace, trailing commas, and detects malformed lines by raising an error.

```
def parse_csv_line(line):
    fields = []
    field = []
    in_quotes = False
    i = 0
    length = len(line)

    while i < length:
        char = line[i]

        if in_quotes:
            if char == '"':
                # Check if this is an escaped quote
                if i + 1 < length and line[i + 1] == '"':
                    field.append('"')
                    i += 1  # skip the escaped quote
                else:
                    # End of quoted field
                    in_quotes = False
            else:
                # Accept any char inside quotes including commas and newlines
                field.append(char)
        else:
```

To handle all these corner cases, you need to implement a stateful parser that processes the line character by character, keeping track of whether you're inside quotes, handling escaped quotes, and recognizing delimiters accordingly.

Here's a robust implementation that considers all the listed corner cases:

```
def parse_csv_line(line):
    fields = []
    field = ''
    in_quotes = False
    i = 0
    length = len(line)

    while i < length:
        char = line[i]

        if char == '"':
            if in_quotes:
                # Check for escaped double quotes ("")
                if i + 1 < length and line[i + 1] == '"':
                    field += '"'
                    i += 1  # Skip the escaped quote
                else:
                    in_quotes = False  # End of quoted field
            else:
                in_quotes = True  # Start of quoted field
        elif char == ',' and not in_quotes:
            # End of field
            fields.append(field)
            if char == ',':
                # End of field
                fields.append(''.join(field))
                field = []
            elif char == '"':
                # Start of quoted field (only valid if field is empty)
                if len(field) == 0:
                    in_quotes = True
                else:
                    # Unexpected quote inside unquoted field: malformed
                    raise ValueError(f'Malformed CSV line: unexpected quote at position {i}')
            else:
                # Normal char outside quotes
                field.append(char)
        i += 1

    # Add the last field
    if in_quotes:
        # Unclosed quote at end of line - malformed
        raise ValueError('Malformed CSV line: unclosed quote at end of line')

    fields.append(''.join(field))

    return fields
```

Explanation

- **Quoted fields** are handled by toggling `in_quotes` when a quote character is encountered.

```
            field = ''
        else:
            # Regular character
            field += char
        i += 1

    # Append the last field
    fields.append(field)

    # Optional: trim whitespace outside quotes if desired
    # For now, preserve as-is to respect spacing inside quotes.
    return fields
```

Explanation:

- **State management:** `in_quotes` tracks whether we're inside a quoted field.
- **Handling quotes:** When encountering a `"`, toggle `in_quotes`, and handle escaped double quotes (`""`) by adding a single `"` to the current field.
- **Handling commas:** Only split at commas when not inside quotes.
- **Unclosed quotes:** This implementation does not explicitly handle unclosed quotes; it will parse until the end. You might want to add validation if needed.
- **Whitespace:** Preserves spaces as-is, but you can strip outside quotes if desired.
- **Empty fields:** Are captured correctly, e.g., consecutive commas.

This implementation correctly handles cases like embedded commas, escaped quotes, empty fields, and more.

- **Escaped quotes** `""` inside quoted fields are translated to a single `"`.
- **Commas inside quotes** are treated as normal characters, not field separators.
- **Empty fields** are handled naturally when encountering `,,`.
- **Trailing commas** add an empty field at the end.
- **Whitespace is preserved** exactly as it appears inside or outside quotes.
- **Malformed lines** (like unclosed quotes or unexpected quotes) raise `ValueError`.

Example usage:

```
line = '"Smith, John",25,"New York"'

print(parse_csv_line(line))

# Output: ['Smith, John', '25', 'New York']



line = '"John ""Johnny"" Smith",30,"NY"'

print(parse_csv_line(line))

# Output: ['John "Johnny" Smith', '30', 'NY']



line = 'John,,"NY",'

print(parse_csv_line(line))

# Output: ['John', '', 'NY', '']



line = '"John\nSmith",30,"NY"'

print(parse_csv_line(line))

# Output: ['John\nSmith', '30', 'NY']
```

If you want, I can help you extend this parser to handle multi-line CSV records as well. Let me know!

Comment	3a: Both outputs deliver comprehensive lists of CSV quoting corner cases—Output 1 catalogs a wide range	3b: Output 1.1 implements a fully stateful parser with escape handling, error detection via exceptions, and

	(escaped quotes, embedded commas, newlines, malformed lines), while Output 2 reorganizes and adds details like line-ending variations and unclosed quotes. Together they thoroughly justify why `split(',')` fails, though there's some overlap between the two lists.	explicit support for every listed edge case—very robust but verbose. Output 2.1 pares this down to a leaner loop-based parser that still handles quoted commas and escapes but skips malformed-line validation, trading completeness for simplicity.

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Repetitions	3	

Prompt	Improve CSV parsing to respect quoted fields.
	Discuss pitfalls, then give finished code.
	
	def parse_csv_line(line):
	    return line.split(',') # broken	

Output	(votes 1/3) When parsing CSV lines, a common pitfall is splitting simply by commas without considering that fields may be quoted and contain commas inside them. For example: John,Doe,"1234, Elm Street",New York If you just split by commas, you get: ["John", "Doe", "\"1234", " Elm Street\"", "New York"] which incorrectly splits the quoted field `"1234, Elm Street"` into two parts. **Pitfalls to watch out for:** 1. **Quoted fields with commas**: commas inside quotes should not be treated as delimiters. 2. **Escaped quotes inside quoted fields**: e.g., `"He said, ""Hello"""` represents a field with embedded quotes. 3. **Empty fields**: consecutive commas may represent empty fields. 4. **Leading/trailing whitespace** can be significant or not depending on CSV dialect. 5. **Newlines inside quoted fields** (usually rare in simple line parsing). The CSV format is actually complicated and has many edge cases, so typically you want to use a CSV parsing	(votes 1/3) Certainly! The current implementation: def parse_csv_line(line): return line.split(',') # broken is naive and does not handle quoted fields, which can contain commas themselves. For example, a line like: John,Doe,"123, Elm Street",New York would be incorrectly split into: ['John', 'Doe', '"123', ' Elm Street"', 'New York'] which is invalid. Proper CSV parsing needs to respect quoted fields, handle embedded commas, and account for escaped quotes within quoted fields. --- **Pitfalls of naive splitting:** - **Embedded commas** inside quoted fields cause incorrect splits. - **Quotes within fields** need to be handled, especially escaped quotes. - **Different CSV dialects** (e.g., different quote characters) might complicate parsing. - **Line breaks within quoted fields** are common

library (like Python's built-in `csv` module). But if you want to implement a basic parser that respects quoted fields, here is an improved version:

Final Answer:

```
def parse_csv_line(line):
    fields = []
    field = ''
    in_quotes = False
    i = 0
    length = len(line)

    while i < length:
        char = line[i]

        if in_quotes:
            if char == '"':
                # Look ahead to check if this is an escaped quote
                if i + 1 < length and line[i + 1] == '"':
                    field += '"'
                    i += 1  # skip the escaped quote
                else:
                    in_quotes = False  # end of quoted field
            else:
                field += char
        else:
            if char == '"':
                in_quotes = True
            elif char == ',':
```

in CSV files, making simple line splitting insufficient.

Improved solution:

Python's built-in `csv` module handles all these issues gracefully. For custom parsing, we can leverage it directly, or implement a state machine to parse lines respecting quotes.

Below is a **finished** implementation using the `csv` module for correctness and simplicity.

Final code:

```
import csv
from io import StringIO


def parse_csv_line(line):
    # Use csv.reader to parse the line respecting quotes
    reader = csv.reader(StringIO(line))
    return next(reader)
```

Usage example:

```
line = 'John,Doe,"123, Elm Street",New York'
fields = parse_csv_line(line)
print(fields)
# Output: ['John', 'Doe', '123, Elm Street', 'New York']
```

This approach correctly handles quoted fields, embedded commas, and escaped quotes, ensuring robust CSV parsing.

```
            fields.append(field)

            field = ''

        else:

            field += char

    i += 1


# Append the last field

fields.append(field)

return fields
```

How this works:

- We iterate over each character.
- When we hit a quote and are not already inside a quoted field, we enter quoted mode.
- Inside quoted mode, commas are treated as normal characters.
- Double quotes inside quoted fields are interpreted as a single literal quote.
- When we encounter a comma outside quotes, we split the field.
- The last field is added after the loop.

This implementation handles quoted commas, escaped quotes, and empty fields correctly.

Example usage:

```
line = 'John,Doe,"1234, Elm Street","He said,
""Hello""",New York'

print(parse_csv_line(line))

# Output:

# ['John', 'Doe', '1234, Elm Street', 'He said, "Hello"',
'New York']
```

This is a minimal, robust CSV line parser respecting quoted fields without using external libraries.

Comment	4a: This solution offers a clear, manual state-machine parser that correctly handles quoted commas, escaped quotes, and empty fields, with a straightforward character loop and split logic—very instructive and robust for single-line parsing. It could be improved by adding error detection for unclosed quotes or support for multi-line fields, but it meets the prompt without external dependencies.	4b: By leveraging Python's built-in `csv` module, this version delivers a one-line, production-grade parser that inherently handles all quoting pitfalls and dialect variations—extremely concise and reliable. However, it skips the custom parsing logic, which may be less educational if you specifically wanted to see a manual implementation.

Model Comparison and Analysis

Under prompt chaining, both models deliver manual state-machine parsers: the mini version (3a) includes exception-based error detection for malformed lines and covers every edge case—very robust but verbose—while the nano version (3b) is leaner, handling quotes and commas correctly but skipping explicit malformed-line checks. In self-consistency mode, the mini model (4a) again opts for the detailed manual parser, preserving educational value, whereas the nano model (4b) switches to Python's `csv` module for a one-liner that's concise and fully RFC-aware but hides the parsing logic.

9. Data Class to API Conversion (Kotlin)

Goal: Convert the data class to a REST API using Ktor:

Zero-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Convert the data class to a REST API using Ktor: data class Product(val id: Int, val name: String, val price: Double) # TODO: Create GET and POST endpoints using Ktor	

Output	Here's an example of how you can create a simple Ktor REST API with GET and POST endpoints for the `Product` data class:
	import io.ktor.application.*
	import io.ktor.features.ContentNegotiation
	import io.ktor.http.*
		Certainly! Here's an example of how to convert your `Product` data class into a REST API using Ktor, with GET and POST endpoints:
		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.embeddedServer
import io.ktor.server.netty.Netty


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


fun main() {
    // In-memory list to store products
    val products = mutableListOf<Product>()


    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            json() // Use kotlinx.serialization or Jackson (depending on your setup)
        }
        routing {
            route("/products") {
                get {
                    // Return all products
                    call.respond(products)
                }
                post {
                    // Receive a product from the request body and add it to the list
                    val product = call.receive<Product>()
                    products.add(product)
                    call.respond(HttpStatusCode.Created,
```

```
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() {
    // Sample in-memory storage for products
    val products = mutableListOf<Product>()


    embeddedServer(Netty, port = 8080) {
        install(ContentNegotiation) {
            json() // Enable JSON serialization/deserialization
        }
        routing {
            route("/products") {
                // GET /products - retrieve all products
                get {
                    call.respond(products)
                }

                // POST /products - add a new product
                post {
                    val product = call.receive<Product>()
                    products.add(product)
```

	product)
	}
	}
	}
	}.start(wait = true)
	}
	Explanation:
	• We use `ContentNegotiation` with `json()` to handle JSON serialization and deserialization.
	• The `/products` GET endpoint returns the current list of products.
	• The `/products` POST endpoint accepts a JSON body representing a `Product`, adds it to the in-memory list, and returns the created product with HTTP 201 status.
	• The server runs on port 8080.
	Make sure you add the necessary dependencies for Ktor and Kotlin serialization or Jackson in your `build.gradle.kts` or `pom.xml`. For example, with Kotlin serialization and Gradle:
	dependencies {
	implementation("io.ktor:ktor-server-netty:2.3.2")
	implementation("io.ktor:ktor-server-content-negotiation:2.3.2")
	implementation("io.ktor:ktor-serialization-kotlinx-json:2.3.2")
	implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.1")
	}
	Let me know if you want me to help with build setup too!
		call.respond(HttpStatusCode.Created, product)
		}
		}
		}
		}.start(wait = true)
		}
		Explanation:
		• **ContentNegotiation** with `json()` enables automatic serialization/deserialization of JSON.
		• **GET /products** returns the list of products.
		• **POST /products** accepts a JSON payload to add a new product.
		How to run:
		Let me know if you'd like additional endpoints or features!Ensure you have Ktor dependencies in your `build.gradle.kts` or `pom.xml`.
		Run the application.
		Use tools like `curl`, Postman, or your browser to test:
		• GET request to `http://localhost:8080/products`
		POST request to `http://localhost:8080/products` with a JSON body, e.g.,
		{
		"id": 1,
		"name": "Sample Product",
		"price": 99.99
		• }

Comment	1a: The implementation is complete and idiomatic—defining GET/POST routes, in-memory storage, and JSON serialization—and even includes dependency snippets and build tips, making it highly	1b: This version mirrors the functionality with clear routing and JSON handling, and adds concise “How to run” instructions for testing; it’s equally correct but less detailed on dependency setup.

	practical though a bit heavy on version specifics.	

Prompt-Chaining

<table>
<tr><td><b>Model</b></td><td>gpt-4.1-mini</td><td>gpt-4.1-nano</td></tr>
<tr><td><b>Temperature</b></td><td colspan="2">0.7</td></tr>
<tr><td><b>Token Limit</b></td><td colspan="2">1024 (512 per response)</td></tr>
<tr><td><b>Prompt 1</b></td><td colspan="2">Sketch the REST routes needed to expose Product via Ktor (URLs, verbs).

data class Product(val id:Int, val name:String, val price:Double)</td></tr>
<tr><td><b>Output 1</b></td><td>Certainly! Here's a typical set of REST routes to expose the <code>Product</code> resource using Ktor, with corresponding HTTP verbs and URL patterns:
<table>
<tr><th>HTTP Verb</th><th>URL</th><th>Description</th></tr>
<tr><td>GET</td><td>/products</td><td>Get all products</td></tr>
<tr><td>GET</td><td>/products/{id}</td><td>Get a product by ID</td></tr>
<tr><td>POST</td><td>/products</td><td>Create a new product</td></tr>
<tr><td>PUT</td><td>/products/{id}</td><td>Update a product by ID</td></tr>
<tr><td>DELETE</td><td>/products/{id}</td><td>Delete a product by ID</td></tr>
</table>
<hr>
<h3>Example Ktor route sketch:</h3>
data class Product(val id: Int, val name: String, val price: Double)

fun Route.productRoutes() {</td><td>Certainly! Here's a typical set of REST routes to expose the <code>Product</code> resource via Ktor, following standard REST conventions:

// Create a new product

POST /products -> create a new product

// Retrieve all products

GET /products -> get list of all products

// Retrieve a specific product by ID

GET /products/{id} -> get product with specified id

// Update an existing product by ID

PUT /products/{id} -> update product with specified id

// Partially update an existing product (optional, often PATCH)

PATCH /products/{id} -> partially update product with specified id

// Delete a product by ID

DELETE /products/{id} -> delete product with</td></tr>
</table>

<table>
<tr>
<td></td>
<td>

```
    route("/products") {
        get {
            // Handle fetching all products
        }
        post {
            // Handle creating a new product
        }
        get("{id}") {
            // Handle fetching a product by id
        }
        put("{id}") {
            // Handle updating a product by id
        }
        delete("{id}") {
            // Handle deleting a product by id
        }
    }
}
```

This setup follows REST conventions and cleanly exposes CRUD operations for `Product`.
</td>
<td>
specified id

Summary:

<table>
<tr><th>HTTP Verb</th><th>URL Pattern</th><th>Action</th></tr>
<tr><td>POST</td><td><code>/products</code></td><td>Create a new product</td></tr>
<tr><td>GET</td><td><code>/products</code></td><td>List all products</td></tr>
<tr><td>GET</td><td><code>/products/{id}</code></td><td>Get a specific product by ID</td></tr>
<tr><td>PUT</td><td><code>/products/{id}</code></td><td>Replace/update a product</td></tr>
<tr><td>PATCH</td><td><code>/products/{id}</code></td><td>Partially update a product</td></tr>
<tr><td>DELETE</td><td><code>/products/{id}</code></td><td>Delete a product</td></tr>
</table>

You can implement these routes in Ktor using routing blocks and handlers for each HTTP verb.
</td>
</tr>
<tr>
<td><b>Prompt 2</b></td>
<td colspan="2">
Turn the sketch below into Kotlin Ktor routing code:

{analysis}
</td>
</tr>
<tr>
<td><b>Output 2</b></td>
<td>
Certainly! Here's a complete Kotlin Ktor routing example implementing the REST endpoints for `Product`. For simplicity, I'll use an in-memory mutable list to simulate a data store.

```
import io.ktor.application.*
```
</td>
<td>
Certainly! Here's how you can implement the REST routes for `Product` in Ktor, assuming you have some kind of in-memory storage or a database:

```
import io.ktor.application.*
```
</td>
</tr>
</table>

```
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.features.ContentNegotiation
import io.ktor.serialization.kotlinx.json.*
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json


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


fun Application.module() {
    install(ContentNegotiation) {
        json(Json { prettyPrint = true })
    }


    val products = mutableListOf<Product>(
        Product(1, "Apple", 0.99),
        Product(2, "Banana", 0.59)
    )


    routing {
        route("/products") {
            // Get all products
            get {
                call.respond(products)
            }
```

```
import io.ktor.http.*
import io.ktor.request.*
import io.ktor.response.*
import io.ktor.routing.*


// Example in-memory storage
val products = mutableMapOf<Int, Product>()
var nextId = 1


fun Application.module() {
    routing {
        route("/products") {
            // Create a new product
            post {
                val product = call.receive<Product>()
                val id = nextId++
                val newProduct = product.copy(id = id)
                products[id] = newProduct
                call.respond(HttpStatusCode.Created, newProduct)
            }


            // Get all products
            get {
                call.respond(products.values.toList())
            }


            // Get product by ID
            get("/{id}") {
```

```kotlin
        // Create a new product
        post {
            val product = call.receive<Product>()
            // Simple check to avoid duplicate IDs
            if (products.any { it.id == product.id }) {
                call.respond(HttpStatusCode.Conflict, "Product with id ${product.id} already exists.")
                return@post
            }
            products.add(product)
            call.respond(HttpStatusCode.Created, product)
        }


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


            val product = products.find { it.id == id }
            if (product == null) {
                call.respond(HttpStatusCode.NotFound, "Product not found")
            } else {
                call.respond(product)
            }
```

```kotlin
            val id = call.parameters["id"]?.toIntOrNull()
            val product = id?.let { products[it] }
            if (product != null) {
                call.respond(product)
            } else {
                call.respond(HttpStatusCode.NotFound, "Product not found")
            }
        }


        // Update (replace) product by ID
        put("/{id}") {
            val id = call.parameters["id"]?.toIntOrNull()
            if (id == null || !products.containsKey(id)) {
                call.respond(HttpStatusCode.NotFound, "Product not found")
                return@put
            }
            val updatedProduct = call.receive<Product>().copy(id = id)
            products[id] = updatedProduct
            call.respond(updatedProduct)
        }


        // Partially update product by ID (optional)
        patch("/{id}") {
            val id = call.parameters["id"]?.toIntOrNull()
            val existing = products[id]
            if (existing == null) {
                call.respond(HttpStatusCode.NotFound, "Product not found")
            }

            // Update a product by ID
            put("{id}") {
                val id = call.parameters["id"]?.toIntOrNull()
                if (id == null) {
                    call.respond(HttpStatusCode.BadRequest, "Invalid or missing id")
                    return@put
                }

                val updatedProduct = call.receive<Product>()

                val index = products.indexOfFirst { it.id == id }
                if (index == -1) {
                    call.respond(HttpStatusCode.NotFound, "Product not found")
                    return@put
                }

                // Optionally check if updatedProduct.id matches path id or ignore it
                if (updatedProduct.id != id) {
                    call.respond(HttpStatusCode.BadRequest, "Product ID in body must match URL path id")
                    return@put
                }

                products[index] = updatedProduct
                call.respond(updatedProduct)
            }
```

```
                    return@patch
                }
                val partial = call.receive<PartialProduct>()
                val updated = existing.copy(
                    name = partial.name ?: existing.name,
                    price = partial.price ?: existing.price
                )
                products[id] = updated
                call.respond(updated)
            }

            // Delete product by ID
            delete("/{id}") {
                val id = call.parameters["id"]?.toIntOrNull()
                if (id == null || products.remove(id) == null) {
                    call.respond(HttpStatusCode.NotFound, "Product not found")
                } else {
                    call.respond(HttpStatusCode.NoContent)
                }
            }
        }
    }
}

// Optional data class for partial updates
data class PartialProduct(val name: String?, val price: Double?)
```

Notes:

- This example uses an in-memory `MutableMap` for storage.

```
            // Delete a product by ID
            delete("{id}") {
                val id = call.parameters["id"]?.toIntOrNull()
                if (id == null) {
                    call.respond(HttpStatusCode.BadRequest, "Invalid or missing id")
                    return@delete
                }

                val removed = products.removeIf { it.id == id }
                if (removed) {
                    call.respond(HttpStatusCode.NoContent)
                } else {
                    call.respond(HttpStatusCode.NotFound, "Product not found")
                }
            }
        }
    }
}
```

Explanation:

- **GET /products** returns the entire list of products.
- **POST /products** accepts a JSON product, checks for duplicate id, and adds it.
- **GET /products/{id}** returns the product with the given id or 404 if not found.
- **PUT /products/{id}** updates an existing product by replacing it with the received product data if the id matches.
- **DELETE /products/{id}** deletes the product with the given id, returning 204 No Content if successful.

Make sure you have Ktor dependencies for `ktor-server-core`, `ktor-server-netty`, and

- The `POST` route assigns a new ID to each product.
- The `PATCH` route uses a `PartialProduct` data class to handle partial updates.
- Error handling is included for missing or non-existent products.

You can adapt the storage and request/response models as needed!

	`ktor-serialization-kotlinx-json` along with Kotlinx Serialization configured. This is a minimal example to get you started with RESTful product CRUD in Ktor!	
Comment	Output 1: Clearly lists all CRUD routes with HTTP verbs and URL patterns, then provides a concise Ktor routing sketch—spot-on for REST conventions and directly actionable. Output 2: Delivers a full-featured Ktor module with JSON serialization, sample data, status-code handling, and validation—very production-ready but somewhat verbose.	Output 1.1: Adds an optional PATCH route and a neat summary table, offering extra clarity on partial updates, though PATCH wasn't requested. Output 2.1: Presents a lean in-memory–backed router with dynamic ID assignment and PATCH support for partial updates—concise and flexible, though it omits explicit serialization setup.

Model Comparison and Analysis

Across zero-shot prompting, GPT-4.1-mini offers a fully idiomatic GET/POST implementation complete with dependency snippets and build tips, while GPT-4.1-nano delivers the same core functionality more succinctly but omits those setup details. In prompt-chaining Prompt 1, the mini model crisply enumerates REST routes and a concise Ktor sketch, whereas the nano model provides a full-featured, production-ready module with JSON serialization and error handling—at the cost of extra verbosity. For Prompt 2 chaining, GPT-4.1-mini enriches the router with an optional PATCH route and a summary table, while GPT-4.1-nano presents a lean in-memory router with dynamic ID assignment and PATCH support but skips explicit serialization setup.

10. Function Summarization (Python)

Goal: Write a brief summary of the function:

Prompt-Chaining

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024 (512 per response)	
Prompt 1	Break down what each line of the Python function does.	

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

Output 1	Certainly! Here's a line-by-line breakdown of the function:
	def reverse_words(sentence):
	• Defines a function named `reverse_words` that takes one parameter, `sentence`, which is expected to be a string.
	return ' '.join(sentence.split()[::-1])
	• `sentence.split()` splits the input string into a list of words using whitespace as the delimiter.
	• `[::-1]` reverses the list of words.
	• `' '.join(...)` joins the reversed list of words back into a single string, with spaces between each word.
	• `return` outputs the resulting reversed-word string from the function.
	Summary:
	The function takes a sentence, splits it into words, reverses the order of the words, and then joins them back into a string with spaces, effectively returning the sentence with the word order reversed.
		Certainly! Here's a line-by-line explanation of the `reverse_words` function:
		def reverse_words(sentence):
		• **Defines a function named `reverse_words`** that takes a single parameter called `sentence` (expected to be a string).
		return ' '.join(sentence.split()[::-1])
		• **`sentence.split()`**:
		  ○ Splits the input string `sentence` into a list of words based on whitespace.
		  ○ For example, `"Hello world"` becomes `["Hello", "world"]`.
		• **`[::-1]`**:
		  ○ This is slicing syntax that reverses the list.
		  ○ So, `["Hello", "world"]` becomes `["world", "Hello"]`.
		• **`' '.join(...)`**:
		  ○ Joins the reversed list of words back into a single string, inserting a space `' '` between each word.
		  ○ Continuing the example, `["world", "Hello"]` becomes `"world Hello"`.
		• **`return`**:
		  ○ The function returns the resulting string with the words in reverse order.
		In summary:
		The function splits the input sentence into words, reverses the order of those words, and then joins them back into a string separated by spaces, effectively reversing the order of words in the sentence.

Prompt 2	Using that breakdown: {analysis} write a one-sentence summary of the function’s purpose.	
Output 2	The function `reverse_words` takes a sentence as input and returns a new sentence with the order of the	The `reverse_words` function takes a sentence as input and returns a new sentence with the order of

	words reversed.	words reversed.
Comment	2a: Output 1 gives a clear, line-by-line mapping from code to behavior and wraps up with a succinct summary—straightforward and easy to follow. 2c: Output 2 delivers a flawless one-sentence summary that captures the function's intent precisely and concisely.	2b: Output 1.1 breaks down each operation in more granular detail (with examples for split, slicing, and join), which is educational but a bit verbose for this simple function. 2d: Output 2.1 echoes the same summary exactly—equally clear but redundant compared to Output 2.

Chain-of-Thought

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Let's break down this function. It splits the sentence into words and then reverses the order of the words. Finally, it joins the reversed words into a string and returns it. Now, summarize its purpose.	

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

Output	The function `reverse_words` takes a sentence as input and returns a new sentence with the order of the words reversed.	The function takes a sentence, reverses the order of its words, and returns the resulting sentence as a string.
Comment	3a: This summary is precise and self-contained—explicitly names the function, its input, and its output—making it immediately clear what `reverse_words` does.	3b: Equally concise and accurate, though it omits the function name, opting for a more generic phrasing; still perfectly captures the core behavior.

Model Comparison and Analysis

Across prompt chaining, GPT-4.1-mini's comments (2a, 2c) stay concise and sharply focused—clearly mapping each code line to behavior and summarizing intent in a single precise sentence—while GPT-4.1-nano's (2b, 2d) veer into extra granularity or redundancy. In chain-of-thought mode, the mini model (3a) again nails specificity—naming the function and its behavior crisply—whereas the nano model (3b) opts for an equally accurate but more generic phrasing. Overall, GPT-4.1-mini consistently delivers the sharper, more informative commentary, while GPT-4.1-nano tends toward extra verbosity or generality.

11. Prompt from Code Comments (Python)

Goal: Write a prompt that could generate the code:

Zero-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Write a prompt that could generate the code:	

```
# This function checks if a number is prime
def is_prime(n):
        if n <= 1:
        return False
        for i in range(2, int(n**0.5) +1):
        if n % i == 0:
        return False
        return True
```

Output	Write a Python function named `is_prime` that takes an integer `n` and returns `True` if `n` is a prime number, and `False` otherwise. The function should efficiently check for factors only up to the square root of `n`. Include a comment at the top describing that the function checks if a number is prime.	Create a Python function named `is_prime` that takes an integer `n` as input and returns `True` if `n` is a prime number, and `False` otherwise. The function should handle numbers less than or equal to 1 by returning `False`, and for larger numbers, it should check divisibility from 2 up to the square root of `n`.
Comment	This prompt is very precise—naming the function, specifying the input/output behavior, instructing to limit checks to √n, and even requesting a top-of-function comment—ensuring the generated code matches the example exactly.	Equally clear on function name, edge-case handling for n≤1, and efficient √n looping, though it omits the explicit request for a descriptive comment at the top.

Few-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Example (prompt from comment) Comment: # add two numbers Prompt: "Write a Python function that adds two numbers."	

	Generate a prompt (one sentence) that could have produced the following prime-check function:

```
# 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	Write a Python function that checks whether a given number is prime.	Write a Python function that determines whether a given number is prime.
Comment	2a: Clear and concise—asks for a Python function to check primality, but omits details on using the √n optimization and adding the descriptive comment.	2b: Equally precise (“determines” vs. “checks” is interchangeable) yet also leaves out efficiency requirements and the top-of-function comment prompt.

Model Comparison and Analysis

Under zero-shot prompting, GPT-4.1-mini (1a) produces a richly detailed instruction—specifying function name, √n optimization, and a top-of-function comment—whereas GPT-4.1-nano (1b) still covers √n checks and n≤1 handling but omits the comment request. In few-shot mode, both GPT-4.1-mini (2a) and GPT-4.1-nano (2b) revert to equally concise, generic prompts (“checks” vs. “determines” primality) that leave out optimization and comment details.

12. Fixing Factorial Bug (Python)

Goal: Fix the bug when input is 0:

Zero-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Fix the bug when input is 0:	

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

Output	The bug is that the loop runs from 1 to n−1 (since	The bug in your code is that the loop runs from 1 up to

	`range(1, n)` excludes `n`), so for `n=0` the loop doesn't run and the function returns `1`, which is correct for `0!`, but for other values it incorrectly computes the factorial. Actually, the bug is that for `n > 1`, the loop runs from 1 to `n-1`, so it computes `(n-1)!` instead of `n!`. To fix the function and handle the input `0` correctly, you should loop from `1` to `n` (inclusive). Also, `0!` is defined as `1`, so returning `1` when `n=0` is correct. Here's the corrected code:	(but not including) n, which causes it to miss multiplying by n itself. Also, for input 0 or 1, the factorial should return 1. Here's the corrected version:

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

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

	This will correctly compute factorials for all `n >= 0`.	This way, when `n` is 0 or 1, it correctly returns 1, and for other positive integers, it calculates the factorial properly.
Comment	1a: The explanation correctly identifies that the loop should include n and the provided fix is accurate, but the initial focus on the n=0 case is misleading—0! was already handled correctly—and the description conflates "handling 0" with the real bug of missing the final multiplication, making the narrative confusing.	1b: This version cleanly pinpoints the missing multiplication by n, explicitly handles the n=0 case, and presents a succinct corrected implementation—very clear and to the point, though the separate 0-case check is technically redundant once the loop is fixed.

Chain-of-Thought

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Think about the issue when n = 0. The loop starts at 1, but it should handle the case where n = 0. Let's fix the loop to handle this case correctly.	

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

Output	You're right! When `n = 0`, the factorial should be `1` by definition, but the current loop `for i in range(1,`	To correctly handle the case when `n = 0`, we should ensure that the loop runs appropriately and that the

	`n):` doesn't run at all (which is fine), but when `n = 1`, it also doesn't run, so the current implementation returns 1 correctly for `n=0` and `n=1`.
	
	However, there is a subtle bug: the loop runs from `1` up to `n-1` because `range(1, n)` excludes `n`. So for `n = 5`, the loop runs with `i` values 1,2,3,4 — missing multiplying by 5.
	
	To fix this, you should use:
	
	`for i in range(1, n + 1):`
	
	`    result *= i`
	
	This will correctly multiply `result` by all numbers from 1 to `n` inclusive.
	
	Here is the corrected function:
	
	`def factorial(n):`
	
	`    result = 1`
	
	`    for i in range(1, n + 1):`
	
	`        result *= i`
	
	`    return result`
	
	This handles the case `n=0` correctly (the loop doesn't run, and `result` stays 1), and for positive integers it computes the factorial properly.
		function returns 1 for `n = 0`, since 0! = 1.
		
		Currently, the loop `for i in range(1, n):` does not include `n` itself and will not run when `n = 0` or `n = 1`.
		
		Here's the corrected version:
		
		`def factorial(n):`
		
		`    result = 1`
		
		`    for i in range(1, n + 1):`
		
		`        result *= i`
		
		`    return result`
		
		Explanation:
		
		• The loop now runs from 1 up to and including `n` (`range(1, n + 1)`).
		• When `n = 0`, the loop doesn't run, and the function correctly returns 1.
		• When `n = 1`, the loop runs once (i=1) and returns 1.
		• For larger `n`, it multiplies all numbers from 1 to `n` as expected.
		
		Example usage:
		
		`print(factorial(0))  # Output: 1`
		
		`print(factorial(5))  # Output: 120`

Comment	2a: This response accurately explains why `range(1, n)` misses multiplying by n, clearly shows the corrected `range(1, n+1)`, and notes that n=0 and n=1 still return 1—but its initial focus on n=0 and n=1 obscures that the real bug affects all n>1.	2b: This version zeroes in on the off-by-one loop issue, presents the single-loop fix cleanly, explains behavior for n=0 and n=1, and even gives usage examples—concise, clear, and well-structured.

Model Comparison and Analysis

Across zero-shot prompting, GPT-4.1-mini (1a) gives a correct fix but buries the real bug behind an undue focus on n=0/1, while GPT-4.1-nano (1b) immediately pinpoints the off-by-one issue and presents a succinct, accurate solution. Under chain-of-thought prompting, the mini model (2a) again provides thorough reasoning—explaining loop bounds and edge cases—but still overemphasizes n=0/1, whereas the nano model (2b) delivers a clear, focused explanation, cleanly fixes the loop, and even illustrates behavior for n=0 and n=1 without extra distraction.

13. Linked List Node Deletion ©

Goal: Implement node deletion by value:

Few-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	

Prompt

Example (C linked-list delete)
Maintain prev & curr pointers; update head if first node matches.

Implement `deleteNode` that removes the node whose data == key:

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

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

Output

gpt-4.1-mini:

Here's a complete implementation of `deleteNode` that removes the first node with `data == key` from a singly linked list, maintaining `prev` and `curr` pointers and updating the `head` if needed:

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


void deleteNode(struct Node** head, int key) {
    if (head == NULL || *head == NULL) return;


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


    // If the head node itself holds the key to be deleted
    if (curr->data == key) {
```

gpt-4.1-nano:

Certainly! Here's a complete implementation of `deleteNode` which traverses the linked list, maintains `prev` and `curr` pointers, and updates the head if the node to delete is the first node:

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


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


void deleteNode(struct Node** head, int key) {
    if (head == NULL || *head == NULL) {
        // Empty list, nothing to delete
        return;
    }
```

```
        *head = curr->next;  // Change head
        free(curr);          // free old head
        return;
    }

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

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

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

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

Explanation:

- Check if the list is empty.
- Check if the head node contains the key; if yes, update `head` and free the node.
- Otherwise, iterate through the list with `prev` and `curr` until you find the node with the key.
- If found, unlink it by adjusting `prev->next`.
- Finally, free the removed node's memory.

```

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

    // Traverse the list to find the node with data == key
    while (curr != NULL && curr->data != key) {
        prev = curr;
        curr = curr->next;
    }

    if (curr == NULL) {
        // Key not found in list
        return;
    }

    // If the node to delete is the head node
    if (prev == NULL) {
        *head = curr->next;
    } else {
        // Bypass the node to delete
        prev->next = curr->next;
    }

    // Free the memory of the deleted node
    free(curr);
}
```

Explanation:

- We first check if the list is empty.
- Use `curr` to traverse the list, and `prev` to keep track of the previous node.
- When the node with `data == key` is found:

		○ If it's the head node (`prev == NULL`), update the head pointer. ○ Otherwise, bypass it by linking `prev->next` to `curr->next`. ● Finally, free the memory allocated for the deleted node to prevent memory leaks. Feel free to ask if you'd like a full example with list creation and testing!
Comment	1a: The implementation cleanly handles empty lists, head-node deletions, and internal-node removals using `prev`/`curr` pointers, correctly frees memory, and the accompanying explanation is structured and concise.	1b: This version is equally correct—checking for an empty list, traversing with `prev`/`curr`, updating the head or bypassing nodes as needed, and freeing memory—while its explanation is a bit more verbose with inline comments and include directives, making it slightly more tutorial-style.

Chain-of-Thought

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	

Prompt

To delete a node in a linked list, we need to find the node with the matching value and remove it. Consider the pointers to previous and next nodes in the list to ensure we correctly unlink the node. Now, implement the node deletion.

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

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

Output

Here's an implementation of `deleteNode` for a singly linked list. The function searches for the node containing the given `key` and deletes it, updating the list pointers accordingly.

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

    if (head == NULL || *head == NULL) return;



    struct Node* temp = *head;
```

Certainly! Here's a complete implementation of the `deleteNode` function that deletes the node with the specified key from a singly linked list:

```
#include <stdio.h>

#include <stdlib.h>



struct Node {
    struct Node* prev = NULL;

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

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

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

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

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

Explanation:

- First, check if the list is empty.
- Then, check if the head node contains the key.
- Otherwise, traverse the list to find the node with the key.
- Keep track of the previous node to update its `next` pointer.
- If the node is found, unlink it and free its memory.

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

void deleteNode(struct Node** head, int key) {
    if (*head == NULL) {
        // List is empty, nothing to delete
        return;
    }

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

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

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

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

    }


    // Unlink the node from linked list

    prev->next = temp->next;



    free(temp); // Free memory

}
```

	- If the key is not found, do nothing.	**Explanation:** - The function takes a pointer to the head pointer (`struct Node** head`) so it can modify the head if needed. - It first checks if the list is empty. - If the head node contains the key, it updates the head to the next node and frees the original head. - Otherwise, it searches for the node with the matching data, keeping track of the previous node. - Once found, it relinks the previous node's `next` to skip over the node to be deleted. - Finally, it frees the memory of the node to prevent memory leaks. This implementation ensures proper deletion even if the node to be deleted is at the beginning, middle, or not present at all.
Comment	2a: Output1 cleanly implements node deletion—checking for an empty list, handling head-node deletion, traversing with `temp`/`prev`, unlinking the node, and freeing its memory—with a concise, stepwise explanation that's easy to follow.	2b: Output2 mirrors the same correct logic but adds `#include` directives and more in-line comments; its explanation is thorough and beginner-friendly, though slightly more verbose than necessary for experienced readers.

Model Comparison and Analysis

Under few-shot prompting, GPT-4.1-mini (1a) delivers a tight, idiomatic deletion function with focused explanation, while GPT-4.1-nano (1b) adds #include directives and more inline comments, making its guidance slightly more verbose. In chain-of-thought mode, the mini model (2a) offers a clear, stepwise breakdown with minimal fluff, whereas the nano model (2b) repeats the same correct logic but layers on extra commentary and setup details, trading brevity for a more tutorial-style presentation.

14. Recursive Function Completion (Python)

Goal: Complete the recursive function for Fibonacci:

Few-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	

Prompt

Example
Base cases: 0→0, 1→1; recurse n-1 + n-2.

Fill in the missing parts of fibonacci(n):

```
def fibonacci(n):
	# TODO
	pass
```

Output

gpt-4.1-mini:

Here is a complete implementation of the Fibonacci function based on your example:

```
def fibonacci(n):

    # Base cases

    if n == 0:

        return 0

    elif n == 1:

        return 1

    # Recursive case

    else:

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

This function returns the nth Fibonacci number, where `fibonacci(0) = 0` and `fibonacci(1) = 1`. For all other values, it recursively computes the sum of the two preceding Fibonacci numbers.

gpt-4.1-nano:

Certainly! Based on the description, the Fibonacci function should handle the base cases where `n` is 0 or 1, and for other values, it should recursively sum the results of the previous two calls.

Here's the complete implementation:

```
def fibonacci(n):

    # Base cases

    if n == 0:

        return 0

    elif n == 1:

        return 1

    # Recursive case

    else:

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

Explanation:

- When `n` is 0, it returns 0.
- When `n` is 1, it returns 1.

		- For any other `n`, it recursively computes `fibonacci(n - 1)` and `fibonacci(n - 2)` and sums them. **Note:** This implementation is straightforward but can be inefficient for large `n` due to repeated calculations. For optimized performance, consider using memoization or iterative approaches.
Comment	1a: Provides a correct, straightforward recursive implementation with clear base cases and a succinct description—ideal for clarity but lacks any discussion of performance.	1b: Mirrors the same correct code but augments it with an explicit explanation of each case and a helpful note on inefficiency and potential memoization—more informative, though slightly more verbose.

Prompt-Chaining

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024 (512 per response)	
Prompt 1	Write the recurrence and base cases for Fibonacci recursion.	

Output 1	Here's the recurrence relation and base cases for the Fibonacci sequence:
	- **Base cases:**
	  - (F(0) = 0)
	  - (F(1) = 1)
	- **Recurrence:**
	  - For (n \geq 2), (F(n) = F(n-1) + F(n-2))
	And here is the corresponding Python code:
		Certainly! Here's the recurrence relation and the base cases for the Fibonacci sequence, along with the implementation:
		Recurrence relation: [F(n) = \begin{cases} 0, & \text{if } n=0 \ 1, & \text{if } n=1 \ F(n-1) + F(n-2), & \text{if } n>1 \end{cases}]
		Implementation:

Prompt 1

```
def fibonacci(n):
    # TODO
    pass
```

Output 1 (gpt-4.1-mini)

```
def fibonacci(n):

    if n == 0:

        return 0  # Base case

    elif n == 1:

        return 1  # Base case

    else:
```

Output 1 (gpt-4.1-nano)

```
def fibonacci(n):

    if n == 0:

        return 0  # Base case

    elif n == 1:

        return 1  # Base case

    else:

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

```
Recursive case
```

This implementation follows the classic recursive approach. Keep in mind that for large n, this can be quite slow due to repeated calculations. For efficiency, consider using memoization or an iterative approach.

Prompt 2	Using them: {analysis} complete the Python function.

Output 2

Certainly! Here's the completed Python function implementing the Fibonacci sequence using the given recurrence and base cases:

```
def fibonacci(n):

    if n == 0:

        return 0  # Base case

    elif n == 1:

        return 1  # Base case

    else:

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

Certainly! Here's the completed version of your Fibonacci function using the recurrence and base cases:

```
def fibonacci(n):

    if n == 0:

        return 0  # Base case

    elif n == 1:

        return 1  # Base case

    else:

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

This implementation directly follows the recurrence relation and base cases you provided. If you'd like a more efficient version, let me know!

Comment	2a: Clearly states base cases and recurrence, then shows matching code—straightforward and correct, though the inline math adds little beyond the code.	2b: Enhances 2a with formal recurrence notation and a performance caveat—more thorough and pedagogical, but a bit overkill for a simple recursive example.
	2c: Provides the exact recursive implementation succinctly following the given recurrence—precise and to the point, though it skips any discussion of efficiency.	2d: Mirrors 2c's code and adds a prompt for optimization suggestions—useful for further refinement but otherwise redundant.

Model Comparison and Analysis

Across few-shot prompting, GPT-4.1-mini (1a) sticks to a direct recurrence-and-code presentation—clean, code-first, minimal commentary—whereas GPT-4.1-nano (1b) layers in formal mathematical notation and a performance caveat, trading brevity for pedagogical depth. In prompt-chaining, the mini model (2a, 2c) again remains succinct—translating the recurrence into working code with just enough explanation—while the nano model (2b, 2d) enriches its outputs with detailed notes on memoization, efficiency, and recursive behavior, making it more verbose but also more educational. In short, mini excels at concise, task-focused answers; nano offers extra context and teaching value at the expense of brevity.

15. Constructor Completion (Python)

Goal: Complete the class constructor:

Zero-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

Complete the class constructor:

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

Output

gpt-4.1:

Certainly! Here’s how you can complete the constructor to accept `name` (required), `age` (required), and `email` (optional, defaulting to `None`):

```
class Person:

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

        self.name = name

        self.age = age

        self.email = email
```

Usage Example:

```
p1 = Person("Alice", 30)

p2 = Person("Bob", 25, "bob@example.com")
```

Let me know if you need further explanation!

Codestral-2501:

To complete the class constructor, you need to add parameters for `name`, `age`, and an optional `email` attribute. Here is how you can do it:

```
class Person:

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

        self.name = name

        self.age = age

        self.email = email



# Example of creating a Person instance

person = Person("John Doe", 30, "john.doe@example.com")

print(person.name)  # Output: John Doe

print(person.age)   # Output: 30

print(person.email) # Output: john.doe@example.com
```

		# Example of creating a Person instance without an email
		person_no_email = Person("Jane Doe", 25)
		print(person_no_email.name) # Output: Jane Doe
		print(person_no_email.age) # Output: 25
		print(person_no_email.email) # Output: None
		In this implementation:
		- The `__init__` method takes three parameters: `name` and `age` (which are required), and `email` (which is optional and has a default value of `None`).
		- The `self.name`, `self.age`, and `self.email` attributes are initialized with the values passed to the constructor.
		This structure allows you to create `Person` objects with a name and age, and optionally with an email address.

Comment	1a: The constructor is implemented cleanly with required `name` and `age` parameters and an optional `email`, plus concise usage examples—straightforward and sufficient for most cases.	1b: This version mirrors the correct constructor and supplements it with instantiation and print examples for both cases (with and without email), making it very beginner-friendly though a bit more verbose.

Few-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	

Prompt	Example
	class Car: def __init__(self,make): self.make=make
	Add params name, age, optional email to the constructor below:
	class Person:
	def __init__(self):
	pass	

Output	Here's how you can add the parameters `name`, `age`, and an optional `email` to the `Person` class constructor: class Person:	class Person: def __init__(self, name, age, email=None):

	def __init__(self, name, age, email=None):
	self.name = name
	self.age = age
	self.email = email
	• `name` and `age` are required parameters.
	• `email` is optional and defaults to `None` if not provided.
		self.name = name
		self.age = age
		self.email = email

Comment	2a: This version not only adds the correct parameters but also explains which are required and which are optional, making it clear and informative.	2b: The implementation is concise and accurate but omits any explanatory context, which may leave beginners guessing about parameter roles.

Model Comparison and Analysis

In zero-shot mode, GPT-4.1 (1a) not only supplies the correct constructor but also explains which parameters are required versus optional, while Codestral-25.01 (1b) delivers only the bare implementation with no contextual notes. Under few-shot prompting, GPT-4.1-mini (2a) again pairs the code with clear guidance on parameter roles, whereas GPT-4.1-nano (2b) sticks to the minimal code snippet, omitting explanatory detail.

16. Binary Search Completion (Java)

Goal: Complete the binary search implementation:

Prompt-Chaining

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024 (512 per response)	

Prompt 1	Explain how classic binary search updates left/right based on arr[mid].
	
	public int binarySearch(int[] arr,int target){
	int left=0,right=arr.length-1;
	while(left<=right){
	int mid=(left+right)/2;
	// TODO
	}
	return -1;
	}	

Output 1

Certainly! In classic binary search, we repeatedly divide the search interval in half. Here’s how we update `left` and `right` based on the value at `arr[mid]`:

- If `arr[mid]` equals the `target`, we've found the element.
- If `arr[mid]` is **less than** the `target`, it means the target (if it exists) is in the **right half** of the array. So, we set `left = mid + 1`.
- If `arr[mid]` is **greater than** the `target`, it means the target is in the **left half**. So, we set `right = mid - 1`.

Here's the code with the update logic:

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

    int left = 0, right = arr.length - 1;

    while (left <= right) {

        int mid = (left + right) / 2; // or int mid = left + (right - left) / 2 to avoid overflow

        if (arr[mid] == target) {

            return mid;

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

            left = mid + 1;

        } else { // arr[mid] > target

            right = mid - 1;

        }

    }

    return -1; // not found

}
```

Summary Table:

Condition	Update
arr[mid] == target	return mid

Certainly! Here is your code with the TODO filled in according to the classic binary search logic:

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

    int left = 0, right = arr.length - 1;

    while (left <= right) {

        int mid = (left + right) / 2;

        if (arr[mid] == target) {

            return mid;

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

            left = mid + 1;

        } else { // arr[mid] > target

            right = mid - 1;

        }

    }

    return -1;

}
```

Explanation:

- If `arr[mid] == target`, return `mid`.
- If `arr[mid] < target`, update `left = mid + 1`.
- If `arr[mid] > target`, update `right = mid - 1`."

arr[mid] < target	left = mid+1
arr[mid] > target	right = mid-1

This approach efficiently narrows down the search space until the target is found or the interval is empty.

Prompt 2

Fill in the TODOs using the explanation:

{analysis}

Output 2

In the classic binary search algorithm, the `left` and `right` pointers are updated based on the comparison of the `target` with the middle element `arr[mid]`. Here's a step-by-step explanation of how these updates are performed:

1. **Initialization**: Start with `left` at the beginning of the array (`0`) and `right` at the end of the array (`arr.length - 1`).
2. **Midpoint Calculation**: Calculate the midpoint `mid` as `(left + right) / 2`.
3. **Comparison**:
    - If `arr[mid]` equals the `target`, the target is found, and you return the index `mid`.
    - If `arr[mid]` is less than the `target`, the target must be in the right half of the array (if it exists). Update `left` to `mid + 1` to search the right half.
    - If `arr[mid]` is greater than the `target`, the target must be in the left half of the array (if it exists). Update `right` to `mid - 1` to search the left half.
4. **Loop Condition**: The loop continues as long as `left` is less than or equal to `right`.
5. **Termination**: If the target is not found, the loop terminates, and you return `-1`.

Here's the complete implementation:

To complete the binarySearch function based on the classic binary search algorithm, you need to fill in the TODO section with the appropriate comparisons and updates to the left and right pointers. Here is the filled-in function:

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

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

        // TODO: Implement the comparison and pointer updates
        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
        }
```

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

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

        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:

1. **Initialization**: `left` is set to `0`, and `right` is set to `arr.length - 1`.
2. **While Loop**: The loop continues as long as `left <= right`.
3. **Midpoint Calculation**: `int mid = (left + right) / 2;`.
4. **Comparison**:
    - If `arr[mid]` equals `target`,

```
    }

    return -1; // Target not found
}
```

Here’s a breakdown of what was added:

Comparison:

If arr[mid] equals target, return mid because the target is found.

If arr[mid] is less than target, it means the target must be in the right half of the array. Update left to mid + 1 to search the right half.

If arr[mid] is greater than target, it means the target must be in the left half of the array. Update right to mid - 1 to search the left half.

By implementing these comparisons and updates, the function correctly narrows down the search range until the target is found or the search range is exhausted, at which point it returns -1 indicating the target is not in the array.

Comment	1a: Thoroughly explains the left/right updates with clear narrative, annotated code, and a summary table—highly instructive, though the table largely restates the code comments. 1b: Offers a detailed, step-by-step rationale followed by the complete implementation, but the lengthy preamble	1c: Neatly fills in the TODO with correct code and provides a succinct explanation of each branch—concise and clear, though it drops the summary table from the first version. 1d: Correctly replaces the TODO and breaks down each comparison, but the explanation feels verbose and partially truncated, making the presentation less

	repeats information and could be more concise.	polished.

Chain-of-Thought

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

Let's write the logic for binary search. First, find the middle index. If the target is smaller than the middle element, adjust the right boundary. If it's larger, adjust the left boundary. Now, implement the binary search.

```
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

Certainly! Here’s how you can implement the logic inside your loop according to your instructions:

- If `arr[mid] == target`, return `mid`.
- If `target < arr[mid]`, adjust `right = mid - 1`.
- If `target > arr[mid]`, adjust `left = mid + 1`.

Here’s the completed method:

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

    int left = 0, right = arr.length - 1;

    while (left <= right) {

        int mid = left + (right - left) / 2; // Avoids potential overflow

        if (arr[mid] == target) {

            return mid;

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

            right = mid - 1;

        } else {
```

Certainly! Below is the complete implementation of the binary search algorithm based on the logic you described:

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

    int left = 0;

    int right = arr.length - 1;



    while (left <= right) {

        int mid = left + (right - left) / 2; // This way of calculating mid avoids overflow



        // Check if target is present at mid

        if (arr[mid] == target) {

            return mid;

        }



        // If target is greater, ignore the left half
```

```
            left = mid + 1;
        }
    }
    return -1;
}
```

Note:
`int mid = left + (right - left) / 2;` is preferred to avoid potential integer overflow when `left` and `right` are large.

Let me know if you'd like an explanation or a recursive version!

```
        if (arr[mid] < target) {
            left = mid + 1;
        } else {
            // If target is smaller, ignore the right half
            right = mid - 1;
        }
    }

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

Explanation:

1. **Initialization**:
    - `left` is set to the beginning of the array (`0`).
    - `right` is set to the end of the array (`arr.length - 1`).
2. **Loop**:
    - The loop continues as long as `left` is less than or equal to `right`.
3. **Mid Calculation**:
    - `mid` is calculated as `left + (right - left) / 2`. This method of calculating the middle index avoids potential overflow issues that can occur with `(left + right) / 2` when `left` and `right` are very large.
4. **Comparison and Adjustment**:
    - If `arr[mid]` equals the `target`, return `mid` since the target is found.
    - If `arr[mid]` is less than the `target`, adjust the `left` boundary to `mid + 1` to ignore the left half.
    - If `arr[mid]` is greater than the `target`, adjust the `right` boundary to `mid - 1` to ignore the right half.
5. **Return**:
    - If the loop exits without finding the target, return `-1` indicating the target is

		not in the array. This implementation ensures that the binary search is performed efficiently with a time complexity of (O(\log n)).
Comment	2a: Output1 cleanly implements the loop logic with an overflow-safe mid calculation, succinctly handles each comparison branch, and even calls out the mid-overflow pitfall—concise and focused.	2b: Output2 mirrors the correct algorithm and pairs it with a comprehensive, step-by-step explanation (including time complexity), making it exceptionally clear though more verbose than strictly necessary.

Model Comparison and Analysis

In prompt-chaining mode, GPT-4.1's comments (1a, 1b) deliver very thorough explanations—complete with narrative, annotated code, and summary tables—but tend to restate themselves and can feel repetitive. Codestral 25.01's prompt-chaining comments (1c, 1d) zero in on the essential code changes and explanations more succinctly, though occasionally at the expense of structure or completeness. In chain-of-thought mode, GPT-4.1 (2a) provides a crisp, focused walkthrough that highlights the overflow-safe mid calculation and comparison logic without extra fluff, whereas Codestral 25.01 (2b) layers on a full step-by-step rationale and complexity notes, trading conciseness for more detailed exposition.

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

Goal: Resolve inconsistency between function name and logic:

Zero-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	
Prompt	Resolve inconsistency between function name and logic:	
Output	Certainly! The inconsistency here is that the **function name** is `isOdd`, which suggests it should return `true` when `x` is **odd**. However, the **logic** inside returns `true` when `x` is **even** (`x % 2 == 0`).	The inconsistency between the function name `isOdd` and its logic can be resolved by correcting the function logic to actually check if the number is odd. Here's the corrected version of the function:

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

	You can resolve this in **two ways**:
	1. Fix the Function Name (if you want to check for even numbers):
	// Returns true if x is even
	bool isEven(int x) {
	    return x % 2 == 0;
	}
	2. Fix the Logic (if you want to check for odd numbers):
	// Returns true if x is odd
	bool isOdd(int x) {
	    return x % 2 != 0;
	}
	Choose one based on your intent.
	 If you want `isOdd(3)` to return `true` and `isOdd(4)` to return `false`, use option 2.
	 If you want `isEven(4)` to return `true` and `isEven(3)` to return `false`, use option 1.
		// Returns true if x is odd
		bool isOdd(int x) {
		    return x % 2 != 0; // Logic now correctly checks if x is odd
		}
		Alternatively, if you intended to create a function that returns `true` if `x` is even, you should rename the function accordingly:
		// Returns true if x is even
		bool isEven(int x) {
		    return x % 2 == 0; // Logic correctly checks if x is even
		}
		Choose the approach that best fits your requirements.

Comment	1a: This output clearly explains the name–logic mismatch, presents both renaming and logic-fixing options, and guides the user on choosing based on intent—very thorough, though offering two fixes might be more than necessary if the intent is obvious.	1b: This version succinctly corrects the logic (and mentions the renaming alternative) in a streamlined way, making it immediately actionable and easy to follow.

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Repetitions	3	

Prompt	Resolve name/logic mismatch in isOdd.
	Reason, then provide corrected C++ code.
	`bool isOdd(int x){`
	`    return x % 2 == 0;`
	`}`	

Output	(votes 1/3) The function is named `isOdd`, which suggests it should return `true` when `x` is an odd number. However, the current implementation returns `true` when `x` is even (`x % 2 == 0`). This is a logic mismatch.
	Correction: Change the condition to check if `x` is odd, i.e., `x % 2 != 0`.
	--- 	
	Corrected C++ code:
	`bool isOdd(int x){`
	`    return x % 2 != 0;`
	`}`
		(votes 1/3) The mismatch occurs because the function `isOdd` returns `true` when `x % 2 == 0`, which actually checks for even numbers, not odd. To correctly identify if a number is odd, the function should return `true` when `x % 2 != 0`.
		Reasoning:
		• The modulus operator `%` returns the remainder of division.
		• For odd numbers, `x % 2` equals 1.
		• For even numbers, `x % 2` equals 0.
		• The function's name `isOdd` suggests it should return `true` for odd numbers, i.e., when `x % 2 != 0`.
		Corrected code:
		`bool isOdd(int x){`
		`    return x % 2 != 0;`
		`}`

Comment	2a: This output succinctly pinpoints the name–logic mismatch and delivers the corrected condition with minimal fuss—clear and to the point, though it could include the full corrected code block for completeness.	2b: Offers the same correct fix but enriches it with a brief explanation of how `%` works for odd vs. even numbers, making the reasoning more explicit—helpful for beginners, though slightly more verbose than necessary.

Model Comparison and Analysis

In zero-shot prompting, GPT-4.1 (1a) delivers a razor-sharp diagnosis—naming the mismatch and offering the corrected condition with minimal commentary—whereas Codestral-25.01 (1b) enriches the fix with a brief primer on how `%` works for odd versus even numbers, making its explanation more explicit but slightly more verbose. Under self-consistency prompting, GPT-4.1-mini (2a) again sticks to a concise, action-focused correction, while GPT-4.1-nano (2b) layers on additional reasoning about the `%` operator and odd/even behavior, trading brevity for extra pedagogical clarity.

18. Prompt Chaining: Bug Identification → Fix (JavaScript)

Goal: Identify and fix the bug:

Zero-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

Identify and fix the bug:

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

Output

Bug Analysis:

The function as written:

```
function isEven(n) {

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

}
```

- **Problem:** `n % 2` returns `0` (for even) or `1` (for odd), but in JavaScript, `0` is falsy and `1` is truthy. So, for `n=4`, `isEven(4)` returns `0`, not `true`.
- The function is expected to return a boolean (`true` or `false`), not a number.

Fix 1: Use strict equality

```
function isEven(n) {

    return n % 2 === 0;

}
```

Fix 2: Use double negation

```
function isEven(n) {

    return !(n % 2);
```

The bug in the `isEven` function is that it returns `1` or `0` instead of `true` or `false`. To fix this, you can explicitly compare the result of `n % 2` to `0`.

Here's the corrected function:

```
function isEven(n) {

    return n % 2 === 0;

}
```

Now, the function will return `true` if `n` is even and `false` if `n` is odd.

	}
	Usage Example:
	console.log(isEven(4)); // true
	console.log(isEven(7)); // false
	Summary:
	Change `return n % 2;` **to** `return n % 2 === 0;`	

Comment	1a: This output thoroughly analyzes the bug, explains JavaScript's truthy/falsy behavior, offers two distinct fixes (strict equality and double negation), includes usage examples, and wraps up with a concise summary—very comprehensive, though a bit lengthy for such a small change.	1b: This version zeroes in on the core issue and fix—explicitly comparing `n % 2` to `0`—in a direct, no-frills way, making it perfect for quick reference but without the additional context or alternative solutions.

Prompt-Chaining

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024 (512 per response)	

Prompt 1	Analyse why JS isEven returns 1|0 not boolean.
	function isEven(n){
	return n % 2;
	}	

Output 1	Great question! Let's analyze the behavior of your function:
	function isEven(n){
	    return n % 2;
	}
	How it works
	• `n % 2` computes the remainder of dividing `n` by 2.
	    ◦ For **even** numbers, the remainder is `0`.
	    ◦ For **odd** numbers, the remainder is `1`.
		Certainly! Based on the analysis, your function should return a **boolean** (`true` for even, `false` for odd) instead of `0` or `1`. Here's the corrected function:
		function isEven(n){
		    return n % 2 === 0;
		}
		Explanation:
		• `n % 2 === 0` evaluates to `true` if the number is even, `false` otherwise.
		• This matches the expected boolean return value

So, the function returns `0` **for even numbers** and `1` **for odd numbers**.

Why isEven returns `1|0` (numbers), not boolean

1. No explicit boolean conversion

- In JavaScript, `0` and `1` are numbers, not booleans.
- The function simply returns the result of `n % 2`—which is always a number.
- It does **not** use a comparison (`==` or `===`) or a Boolean conversion (like `!!`).

2. How to return a boolean

If you want a boolean (`true` for even, `false` for odd), you could do:

```
function isEven(n) {

    return n % 2 === 0;

}
```

or

```
function isEven(n) {

    return !(n % 2);

}
```

3. Summary Table

Input	`n % 2`	isEven(n) (yours)	Boolean expectation
2	0	0	true
3	1	1	false

In summary:

Your `isEven` function returns `1` or `0` (numbers), not booleans, because it simply returns the result of `n % 2` without converting the result into a boolean using comparison (`=== 0`) or logical operators (`!` or `!!`). In

for a function named `isEven`.

Example usage:

```
console.log(isEven(2)); // true

console.log(isEven(3)); // false
```

Summary:
Replace

return n % 2;

with

return n % 2 === 0;

to make `isEven` return a boolean value as expected.

	JavaScript, returning a number will not automatically convert it to a boolean.
	To make `isEven` return a boolean, use:
	function isEven(n) {
	return n % 2 === 0;
	}
	Let me know if you want more examples or clarification!	

Prompt 2	Fix the function based on that analysis: {analysis}	

Output 2	In JavaScript, the function `isEven` is designed to return an integer value rather than a boolean. Specifically, the expression `n % 2` calculates the remainder when `n` is divided by 2. Here's a detailed breakdown of why `isEven` returns 1 or 0 instead of a boolean:
	1. **Modulus Operator**: The modulus operator `%` returns the remainder of the division of one number by another. When `n` is divided by 2:
	○ If `n` is even, the remainder is 0.
	○ If `n` is odd, the remainder is 1.
	2. **Returning Integer Values**:
	○ If `n % 2` equals 0, the function returns 0, indicating that `n` is even.
	○ If `n % 2` equals 1, the function returns 1, indicating that `n` is odd.
	This approach leverages the fact that `0` is falsy and any non-zero number (like 1) is truthy in JavaScript. However, the function as written does not explicitly convert this result to a boolean. Instead, it returns the integer value directly.
	Advantages of Returning Integers:
	1. **Clarity**: The integer result is clear and straightforward. It explicitly tells you whether the number is even (0) or odd (1).
	2. **Direct Use**: In some contexts, the integer result might be more useful than a boolean. For example, you might want to use the result in
		To fix the function to return a boolean value instead of an integer, you can use the comparison operator === to check if the remainder (n % 2) is equal to 0. This will return true if n is even and false if n is odd.
		Here is the updated function:
		function isEven(n) {
		return n % 2 === 0; // Returns true if n is even, false otherwise
		}
		Example Usage:
		console.log(isEven(4)); // true
		console.log(isEven(7)); // false
		Explanation:
		n % 2 === 0: This expression checks if the remainder

	further calculations or as an index in an array.
	Converting to Boolean:
	If you need a boolean result, you can easily convert the integer to a boolean using the `!!` operator (double negation) or by comparing the result to `0`.
	function isEven(n) {
	return n % 2 === 0; // Returns true if n is even, false otherwise
	}
	Example Usage:
	console.log(isEven(4)); // true
	console.log(isEven(7)); // false
	Summary:
	The `isEven` function returns 1 or 0 because it uses the modulus operator to determine the remainder of the division by 2. This integer result can be interpreted directly (even or odd) without needing to convert it into a boolean. If you prefer a boolean return value, you can adjust the function accordingly.
		of n divided by 2 is equal to 0.
		If n is even, the remainder is 0, so the expression evaluates to true.
		If n is odd, the remainder is 1, so the expression evaluates to false.
		This way, the function isEven will return a boolean value indicating whether the number n is even or not.

Comment	2a: Output 1 offers a deep dive into why `isEven` returns `0`/`1` instead of booleans—covering `%` behavior, JavaScript's truthy/falsy semantics, and even a summary table—making it very thorough, though its length and table feel a bit overkill for the core insight. 2b: Output 2 continues with an exhaustive rationale—outlining the integer return's potential benefits, conversion techniques, and examples before finally showing the boolean-based fix—highly informative but somewhat meandering relative to the simple bug at hand.	2c: Output 1.1 cuts straight to the chase with a one-line corrected function, brief explanation, and usage examples—concise and practical, though devoid of the richer context from the initial analysis. 2d: Output 2.1 mirrors 1.1's succinct fix and example usage, delivering a crystal-clear corrected function; it's optimally focused but largely redundant given the previous corrected snippet.

Model Comparison and Analysis

Across zero-shot prompting, GPT-4.1 (1a) delivers a richly detailed breakdown—covering `%` behavior, truthy/falsy semantics, two fix options, usage examples, and a summary—whereas Codestral-25.01 (1b) zeroes in immediately on the core issue with a single, direct strict-equality fix. Under prompt-chaining, GPT-4.1 (2a, 2b) again provides deep rationale—reiterating operator mechanics, alternative approaches, and examples—at the cost of verbosity, while Codestral-25.01 (2c, 2d) trades depth for brevity, giving clean one-line fixes and minimal commentary but skipping broader context or error-handling guidance.

19. Summary Decomposition (C++)

Goal: Decompose the high-level comment/summary into logical steps:

Prompt-Chaining

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt 1	Decompose the high-level comment into ordered steps the function performs.
	// Function that validates an input, calculates square, and returns result
	int process(int x){
	if(x<0) return -1;
	return x*x;
	}	

Output 1	Certainly! Here are the ordered steps the function performs, as decomposed from the high-level comment: 1. **Validate the input:** Check if the input `x` is less than 0. ○ If `x` is negative, return -1 to indicate invalid input. 2. **Calculate the square:** If the input is valid (i.e., `x` is not negative), compute the square of `x` (i.e., `x * x`). 3. **Return the result:** Return the computed square as the function's result.	Certainly! Here is the ordered list from the analysis: 1. Validate the input by checking if `x` is less than 0. 2. If `x` is negative, return -1. 3. If `x` is non-negative, calculate the square of `x`. 4. Return the computed square as the result.
Prompt 2	Return the ordered list from analysis: {analysis}	
Output 2	Certainly! Let's break down the high-level comment into ordered steps that the function performs: 1. **Validate Input**: Check if the input `x` is negative. ○ If `x` is less than 0, return `-1`.	The ordered list of steps the function performs is as follows: 1. Validate Input: Check if the input x is negative. 2. Calculate Square: If the input x is not negative, calculate the square of x. 3. Return Result: Return the calculated square

<table>
<tr>
<td></td>
<td>

2. **Calculate Square**: If the input `x` is not negative, calculate the square of `x`.
   - Compute `x * x`.
3. **Return Result**: Return the calculated square value.

Here is the function with these steps explicitly outlined:

<pre>
int process(int x) {
    // Step 1: Validate Input
    if (x < 0) {
        return -1;
    }

    // Step 2: Calculate Square
    int result = x * x;

    // Step 3: Return Result
    return result;
}
</pre>

This decomposition makes the function's logic clear and easy to understand.

</td>
<td>value.</td>
</tr>
<tr>
<td>

Comment

</td>
<td>

1a: Output 1 clearly lists each step—validation, error return, calculation, and result return—in order, but the formatting with extra blank lines makes it feel slightly verbose.

1b: Output 2 not only breaks down the steps but also injects them back into the code with comments, making it immediately actionable, though the explanation repeats what the code already shows.

</td>
<td>

1c: Output 1.1 offers a tight, bullet-style list of steps without extra spacing—concise and easy to scan, though it omits the “return -1” detail’s context.

1d: Output 2.1 distills the process into three clear, labeled steps—very succinct—but doesn’t reintegrate them into code, which may reduce its direct usability.

</td>
</tr>
</table>

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	

Token Limit	1024	
Repetitions	3	
Prompt	Break down the C++ function’s behaviour into atomic steps. List them clearly.	

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

Output	(votes 1/3) 1. Receive an integer input `x`. 2. Check if `x` is less than 0. 3. If `x` is less than 0, return -1 immediately. 4. Otherwise, calculate the square of `x` (i.e., `x * x`). 5. Return the calculated square value.	(votes 1/3) ● Receive input parameter `x`. ● Check if `x` is less than 0. ● If `x` is less than 0, return `-1`. ● If `x` is greater than or equal to 0, calculate `x * x`. ● Return the calculated square value.
Comment	2a: This list is clear and accurate—each atomic step is covered in order, with concise wording and the immediate return on invalid input highlighted effectively.	2b: Nearly identical in correctness, though it uses “greater than or equal to 0” instead of “otherwise,” which is slightly more verbose but equally clear.

Model Comparison and Analysis

GPT-4.1’s prompt-chaining outputs (1a/1b) deliver crisp, bullet-style step lists with just enough detail—“otherwise” covers the non-negative case—whereas Codestral 25.01 (1c/1d) matches accuracy but leans on slightly more explicit phrasing (“greater than or equal to 0”), trading a bit of brevity for clarity. In self-consistency mode, GPT-4.1-mini (2a) and GPT-4.1-nano (2b) both produce virtually identical, succinct atomic breakdowns, showing that for straightforward enumeration tasks the mini and nano variants perform equally well with clear, direct step listings.

20. Purpose Inference → Completion (Python)

Goal: Complete the function based on intent:

Chain-of-Thought

Model	gpt-4.1	Codestral-2501
Temperature	0.7	

Token Limit	1024

Prompt

The function is likely intended to calculate an average of a list of scores. We need to sum the scores and then divide by the total number of scores. Complete the function to return the average.

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

Output

Certainly! Here’s how you can complete the function:

```
def calculate_average(scores):
    total = 0
    for score in scores:
        total += score
    if len(scores) == 0:   # avoid division by zero
        return 0
    return total / len(scores)
```

Or more simply:

```
def calculate_average(scores):
    if not scores:   # handles empty list
        return 0
    return sum(scores) / len(scores)
```

Both approaches will work!

To complete the function `calculate_average` so that it returns the average of a list of scores, you need to sum the scores and then divide by the total number of scores. Here's how you can do it:

```
def calculate_average(scores):
    total = 0
    if len(scores) == 0:
        return 0  # To handle the case of an empty list

    for score in scores:
        total += score

    average = total / len(scores)
    return average


# Example usage:
scores = [90, 85, 88, 92, 87]
average = calculate_average(scores)
print("The average score is:", average)
```

Explanation:

1. **Initialize `total`:** Set `total` to 0 to start accumulating the sum of the scores.
2. **Check for empty list:** If the list `scores` is empty, return 0 to avoid division by zero.
3. **Sum the scores:** Use a `for` loop to iterate through each score in the list and add it to `total`.
4. **Calculate the average:** Divide `total` by the number of scores (which is `len(scores)`).

		5. **Return the average:** Return the calculated average. This function will correctly compute and return the average of the scores provided in the list.
Comment	1a: Provides both a loop-based and a one-liner `sum()/len()` solution, explicitly guards against division-by-zero, and is concise and Pythonic—excellent for quick adoption.	1b: Delivers a correct implementation with an example usage and a detailed, step-by-step explanation—very instructive for beginners, though more verbose than needed for seasoned developers.

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Repetitions	3	

Prompt	Infer intent (average of scores) and finish implementation.
	Walk through formula, then code.
	
	def calculate_average(scores):
	    total = 0
	    # TODO
	    pass
	<completed calculate_average>	

Output	(votes 1/3) Let's analyze the intent and then complete the implementation step by step. **Intent Explanation** The function `calculate_average` takes a list of scores and returns their average. The average is calculated as: [\text{average} = \frac{\sum \text{scores}}{\text{number of scores}}] **Walkthrough of the Formula** 1. **Sum all the scores:**	(votes 1/3) Let's walk through how to compute the average of a list of scores. **Understanding the formula:** The average (mean) of a set of numbers is calculated as: [\text{average} = \frac{\text{sum of all scores}}{\text{number of scores}}] In Python, this translates to summing all elements in the list and dividing by the length of the list. **Implementation:** def calculate_average(scores): total = 0

Initialize `total = 0`.
For each score in the list, add it to `total`.

2. **Count the number of scores:**

   Use the length of the list: `len(scores)`

3. **Calculate the average:**

   Divide the `total` by the length of the list.

4. **Handle edge cases:**

   If the list is empty, avoid division by zero; return 0 or handle appropriately.

Code Implementation

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

Final Answer:

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

```
    for score in scores:
        total += score
    average = total / len(scores) if scores else 0
    return average
```

Explanation:

- Initialize `total` to 0.
- Loop through each `score` in the `scores` list, adding it to `total`.
- After the loop, calculate `average` by dividing `total` by the number of scores (`len(scores)`).
- Handle the case where `scores` might be empty to avoid division by zero; in that case, return 0.
- Return the computed average.

Final code:

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

Comment	2a: This output excels at teaching—laying out the mathematical formula in LaTeX, walking through each step (sum, count, divide, edge-case), and then presenting the code. It's highly educational, though the narrative is a bit long for such a simple function.	2b: This version strikes a nice balance—explaining the formula in plain language, showing a compact implementation with an inline empty-list check, and keeping the code minimal. It's clear and concise, though it omits the deeper walkthrough of individual steps.

Model Comparison and Analysis

Under chain-of-thought prompting, GPT-4.1 (1a) delivers a deep, pedagogical walkthrough—complete with formal formula notation and stepwise reasoning—at the expense of brevity, whereas Codestral 25.01 (1b) provides a leaner narrative that focuses on the core formula and code without extra narrative layers. In self-consistency mode, GPT-4.1-mini (2a) again leans into a full educational breakdown, while GPT-4.1-nano (2b) opts for a succinct inline explanation and compact implementation, sacrificing some depth for clarity and conciseness.

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

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

Few-Shot

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	
Prompt	Example (refactor) Add input validation, use pathlib, guard main, etc. Review the utility script below, list issues, then provide a refactored version:	

```
# 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	**Issues in the original script:**
	1. **No Input Validation**:
	○ `filepath` is used directly; no check if file exists or is readable.
	○ `index` is not validated for type, range, or if the column exists in all rows.
	○ No check if CSV is empty or has headers.
	○ No error handling for non-numeric values in the column.
	2. **No Use of `pathlib`**:
	○ Uses string paths and `open()` instead of the more robust `pathlib.Path`.
	3. **Unsafe File Handling**:
	○ Opens file without specifying encoding.
	○ No error handling for file reading.
	4. **Poor Function/Variable Naming & Formatting**:
	○ No PEP8 formatting; statements crammed on lines, poor spacing.
	○ Variables like `data`, `values` are generic.
	5. **Hard-coded File Path and Column Index**:
	○ No way to specify input file or column via command-line arguments.
	6. **No Docstrings**:
	○ Functions lack documentation.
	7. **No Type Hints**:
	○ No type hints for functions.
	8. **No Logging or User Feedback for Errors**:
	○ Script fails silently or with traceback if input is bad.
	--- 	
	Refactored Version:
	import csv
	import sys
	from pathlib import Path
		Sure, let's review the utility script and list the issues:
		1. **Input Validation**: The script does not validate the `filepath` or check if the file exists.
		2. **Path Handling**: It uses a string for the file path instead of a `pathlib.Path` object.
		3. **Main Guard**: The `main` function is not protected by a guard clause.
		4. **Error Handling**: There is no error handling for file operations or data processing.
		Now, let's refactor the script to address these issues:
		# utils.py
		import csv
		import sys
		from pathlib import Path
		def read_csv(filepath):
		if not filepath.exists() or not filepath.is_file():
		raise FileNotFoundError(f"The file {filepath} does not exist.")
		with filepath.open('r') as f:
		return [row for row in csv.reader(f)]
		def summarize_column(data, index):
		if not data:
		raise ValueError("Data is empty")
		try:
		values = [float(row[index]) for row in data[1:]]
		except (ValueError, IndexError) as e:
		raise ValueError(f"Error processing data at index

```
from typing import List, Tuple, Any, Optional


def read_csv(filepath: Path) -> List[List[str]]:
    """
    Reads a CSV file and returns its contents as a list of rows.
    Raises FileNotFoundError if file does not exist.
    Raises csv.Error if file is not a valid CSV.
    """
    if not filepath.exists():
        raise FileNotFoundError(f"File not found: {filepath}")
    if not filepath.is_file():
        raise ValueError(f"Path is not a file: {filepath}")

    with filepath.open('r', encoding='utf-8') as f:
        reader = csv.reader(f)
        data = [row for row in reader]
    if not data:
        raise ValueError(f"CSV file is empty: {filepath}")
    return data


def summarize_column(data: List[List[str]], index: int) -> Tuple[float, float]:
    """
    Summarizes a numeric column by index: returns (total, average).
    Skips rows where conversion to float fails.
    Raises ValueError if no valid numeric data in the column.
```

```
{index}: {e}")

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


def main():
    if len(sys.argv) != 2:
        print("Usage: python utils.py <path_to_csv>")
        sys.exit(1)

    filepath = Path(sys.argv[1])

    try:
        data = read_csv(filepath)
    except FileNotFoundError as e:
        print(e)
        sys.exit(1)

    try:
        total, avg = summarize_column(data, 1)
    except ValueError as e:
        print(e)
        sys.exit(1)

    print("Total:", total)
    print("Average:", avg)


if __name__ == '__main__':
    """
    if not data or len(data) < 2:
        raise ValueError("CSV must have at least a header and one data row.")
    if not isinstance(index, int) or index < 0 or index >= len(data[0]):
        raise IndexError(f"Column index {index} out of range for data with {len(data[0])} columns.")

    values = []
    for row_num, row in enumerate(data[1:], start=2):  # skip header
        try:
            value = float(row[index])
            values.append(value)
        except (ValueError, IndexError):
            print(f"Warning: Skipping invalid or missing value on row {row_num}: {row}", file=sys.stderr)
            continue

    if not values:
        raise ValueError(f"No valid numeric data found in column {index}.")

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


def main():
    import argparse
```

```
    main()
```

Key Changes:

1. **Input Validation**:
    - Added a check to ensure the file exists and is a file using `pathlib.Path`.
2. **Path Handling**:
    - Used `pathlib.Path` for handling the file path.
3. **Main Guard**:
    - Protected the `main` function with a standard guard clause.
4. **Error Handling**:
    - Added error handling for file operations and data processing.
5. **Command-Line Argument Handling**:
    - Used `sys.argv` to accept the file path from the command line.

This refactored version ensures the script is more robust, user-friendly, and easier to maintain.

```
    parser = argparse.ArgumentParser(description="Summarize a numeric column in a CSV file.")

    parser.add_argument('filepath', type=Path, help="Path to the CSV file.")

    parser.add_argument('-c', '--column', type=int, default=1, help="Zero-based column index to summarize (default: 1).")

    args = parser.parse_args()



    try:

        data = read_csv(args.filepath)

        total, avg = summarize_column(data, args.column)

        print(f"Total: {total}")

        print(f"Average: {avg}")

    except Exception as e:

        print(f"Error: {e}", file=sys.stderr)

        sys.exit(1)




if __name__ == '__main__':

    main()
```

Key Improvements:

- Uses `pathlib.Path` for file handling.
- Adds type hints.
- Input validation for file existence and column index.
- Graceful error handling and user feedback.
- Command-line interface for file path and column index.
- Skips (and warns about) non-numeric or missing values.
- PEP8-compliant formatting and informative docstrings.

Comment	1a: This refactor is extremely thorough—leveraging	1b: The second version succinctly addresses the core

	`pathlib`, type hints, detailed docstrings, comprehensive input validation, error handling with warnings, and a full-fledged CLI via `argparse`—making it highly production-ready, though its verbosity may be overkill for simpler scripts.	issues—adds `pathlib` validation, a main guard, and targeted error handling—resulting in a lean, maintainable script, but it skips advanced touches like type hints, docstrings, and a more flexible CLI.

Self-Consistency

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Repetitions	3	

Prompt

Review util script: list issues (I/O, errors), then provide clean refactor.

```
# utils.py - Script to parse and summarise numeric CSV files
import csv
def read_csv(filepath):
        with open(filepath,'r') as f:
        return [row for row in csv.reader(f)]
def summarize_column(data,index):
        values=[float(row[index]) for row in data[1:]]
        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()
<refactored utils.py>
```

Output

(votes 1/3)

Issues in the provided script

I/O Issues:

- No error handling on file open/read (file may not exist or be unreadable).
- Assumes file encoding and format without specifying (may cause issues with non-UTF-8 or malformed CSV).
- Hardcoded filename in `main()` limits reusability.

Error Handling Issues:

- No handling for empty files or missing data (e.g., empty rows, missing columns).
- No validation for converting strings to float; will

(votes 1/3)

```
# utils.py - Script to parse and summarise numeric CSV files


import csv


def read_csv(filepath):

  """Reads CSV file and returns list of rows."""

  with open(filepath, 'r', newline='') as f:

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

raise `ValueError` if non-numeric data is present.
- No handling for division by zero if column has no numeric data.
- No user feedback or logging for errors.

Code Style & Structure:

- Multiple statements on one line reduce readability.
- Function names and variables could be more descriptive.
- No docstrings or comments.
- No command-line argument support.
- No separation of concerns or modularity for better testing.

Refactored `utils.py`

```
"""

utils.py - Script to parse and summarize numeric CSV files.

"""



import csv

import sys



def read_csv(filepath):

    """

    Reads a CSV file and returns the data as a list of rows.

    Each row is a list of strings.



    Raises FileNotFoundError or IOError on file access issues.

    """

    with open(filepath, 'r', encoding='utf-8') as f:

        return list(csv.reader(f))



def summarize_column(data, index):

    """Calculates total and average of a numeric column specified by index."""

    # Extract and convert column values to float, skip header

    try:

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

    except (IndexError, ValueError) as e:

        raise ValueError(f"Error processing column {index}: {e}")

    total = sum(values)

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

    return total, avg



def main():

    filepath = 'data.csv'

    data = read_csv(filepath)

    total, avg = summarize_column(data, 1)

    print(f"Total: {total}")

    print(f"Average: {avg}")



if __name__ == '__main__':

    main()
```

Issues addressed:

- **I/O Error Handling:** Added `newline=''` in `open()` for better CSV compatibility across platforms.
- **Parsing Errors:** Wrapped float conversion in try-except to catch invalid data.
- **Code Readability:** Used consistent spacing, added docstrings, formatted output with f-strings.
- **Robustness:** Checked for empty data before dividing for average to prevent ZeroDivisionError.

```
def summarize_column(data, index):
    """
    Calculates the sum and average of numeric values in the specified column index.

    Args:
        data: List of rows (including header).
        index: Integer column index to summarize.

    Returns:
        (total, average) tuple of floats.

    Raises:
        IndexError if index is out of range.
        ValueError if conversion to float fails.
        ZeroDivisionError if no numeric data found.
    """
    values = []
    for row_num, row in enumerate(data[1:], start=2):  # skip header, keep track of line number
        try:
            value = float(row[index])
            values.append(value)
        except IndexError:
            raise IndexError(f"Row {row_num} does not have column index {index}.")
        except ValueError:
            # Skip or raise? Here we choose to skip non-numeric with a warning
            print(f"Warning: Non-numeric data in row {row_num}, column {index}: '{row[index]}'; skipping.",
```

- **Maintainability:** Clear function separation and comments for clarity.

```
file=sys.stderr)
            continue

    if not values:
        raise ZeroDivisionError("No numeric data found in the specified column.")

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

def main():
    import argparse

    parser = argparse.ArgumentParser(description="Summarize a numeric column from a CSV file.")
    parser.add_argument('filepath', help='Path to the CSV file')
    parser.add_argument('column', type=int, help='Zero-based index of the column to summarize')
    args = parser.parse_args()

    try:
        data = read_csv(args.filepath)
    except FileNotFoundError:
        print(f"Error: File '{args.filepath}' not found.", file=sys.stderr)
        sys.exit(1)
    except IOError as e:
        print(f"Error reading file '{args.filepath}': {e}", file=sys.stderr)
        sys.exit(1)

    if not data:
        print("Error: CSV file is empty.", file=sys.stderr)
        sys.exit(1)

    try:
        total, average = summarize_column(data, args.column)
    except IndexError as e:
        print(f"Error: {e}", file=sys.stderr)
        sys.exit(1)
    except ZeroDivisionError as e:
        print(f"Error: {e}", file=sys.stderr)
        sys.exit(1)

    print(f"Total: {total}")
    print(f"Average: {average}")

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

Summary of improvements

- Added argument parsing for flexible file and column input.
- Added file I/O error handling.
- Added error handling for missing or invalid data.
- Added warnings for non-numeric values instead of failing immediately.
- Added docstrings and comments.
- Improved code readability and style (no multiple statements per line).
- Printed errors to stderr.
- Added encoding to `open()` to avoid locale issues.
- Program exits with appropriate error codes on

	failure.	
Comment	2a: This refactor is extremely comprehensive—adding argument parsing, robust I/O validation, error handling with clear stderr messages, docstrings, and improved style—making it production-ready, though its verbosity might be overkill for simpler use cases.	2b: The second version succinctly tightens the original script—introducing CSV newline handling, basic float-conversion error catching, and docstrings—resulting in a lean, readable utility, albeit without flexible CLI support or full I/O/error validation.

Model Comparison and Analysis

In few-shot mode, GPT-4.1 delivers a full-featured, production-ready refactor—adding CLI parsing, detailed I/O checks, docstrings, and type hints—whereas Codestral-25.01 opts for a leaner overhaul that targets core I/O and error cases without extra bells and whistles. Under self-consistency, GPT-4.1-mini again produces the most exhaustive version with comprehensive validation and user feedback, while GPT-4.1-nano trades off that depth for a succinct, focused refactor that improves compatibility and basic error handling with minimal overhead.

22. Code Completion and Robustness Enhancement (Python)

Goal: Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly.

Zero-Shot

Model	gpt-4.1-mini	gpt-4.1-nano
Temperature	0.7	
Token Limit	1024	
Prompt	Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly:	

```
# file_processor.py - Incomplete script for processing text files

import string

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

def clean_line(line):
        # TODO: Remove punctuation and make lowercase
```

```
    pass

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

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

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

Output

Here's the completed `clean_line` function that removes punctuation and converts the text to lowercase:

```
def clean_line(line):

  # Remove punctuation using str.translate and make lowercase

  return line.translate(str.maketrans('', '', string.punctuation)).lower()
```

Full updated script:

```
import string



def load_file(filepath):

  with open(filepath, 'r') as f:

    return f.readlines()



def clean_line(line):

  # Remove punctuation and make lowercase

  return line.translate(str.maketrans('', '', string.punctuation)).lower()
```

Certainly! Here's the completed script with the `clean_line` function properly implemented to remove punctuation and convert text to lowercase:

```
# file_processor.py - Incomplete script for processing text files



import string



def load_file(filepath):

  with open(filepath, 'r') as f:

    return f.readlines()



def clean_line(line):

  # Remove punctuation and convert to lowercase

  translator = str.maketrans('', '', string.punctuation)

  cleaned_line = line.translate(translator).lower()

  return cleaned_line



def count_words(lines):
```

```
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()
```

This will clean each line by removing punctuation and converting it to lowercase before counting words.

```
    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 changes:

- In `clean_line`, I created a translation table that maps all punctuation characters to `None`, effectively removing them from the string.
- The line is then converted to lowercase with `.lower()`.
- This ensures that words are cleaned consistently, and punctuation does not interfere with word counting.

This script will now read `input.txt`, clean each line, split into words, and count their frequencies accurately.

Comment	1a: This refactor is extremely thorough—leveraging `pathlib`, type hints, detailed docstrings, comprehensive input validation, error handling with warnings, and a full-fledged CLI via `argparse`—making it highly production-ready, though its verbosity may be overkill for simpler scripts.	1b: The second version succinctly addresses the core issues—adds `pathlib` validation, a main guard, and targeted error handling—resulting in a lean, maintainable script, but it skips advanced touches like type hints, docstrings, and a more flexible CLI.

Chain-of-Thought

Model	gpt-4.1	Codestral-2501
Temperature	0.7	
Token Limit	1024	

Prompt

Let's clean up the lines by removing punctuation and converting to lowercase. Then, we'll split the lines into words and count their frequencies. Complete the script to handle file processing and word counting.

```
# file_processor.py - Incomplete script for processing text files

import string

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

def clean_line(line):
        # TODO: Remove punctuation and make lowercase
        pass

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

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

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

Output

gpt-4.1:

Here's the completed script with `clean_line()` implemented to remove punctuation and convert to lowercase:

```
import string



def load_file(filepath):

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

Codestral-2501:

Sure! I'll complete the script to handle file processing and word counting by implementing the `clean_line` function. Here's the full script:

```
# file_processor.py - Complete script for processing text files



import string
```

```
        return f.readlines()


def clean_line(line):
    # Remove punctuation using str.translate, then lowercase
    translator = str.maketrans('', '', string.punctuation)
    return line.translate(translator).lower()


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


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


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

Explanation:

- `clean_line()` uses `str.translate` to strip punctuation and `str.lower()` to lowercase the line.
- The rest of the logic remains unchanged, now operating on cleaned lines.
- The word counting and sorting/printing will work

```
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)
    return line.translate(translator).lower()


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


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


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

Explanation:

1. **load_file(filepath)**: This function reads all lines from the file specified by `filepath` and returns

	as expected.	them as a list of strings. 2. **clean_line(line)**: This function takes a line of text, removes all punctuation using `str.translate` and `str.maketrans`, and converts the entire line to lowercase. 3. **count_words(lines)**: This function processes each line to clean it and then splits it into words. It counts the frequency of each word using a dictionary. 4. **main()**: This is the main function that ties everything together. It loads the file, counts the words, and prints the word counts in sorted order. **Usage:** - Make sure you have an `input.txt` file in the same directory as the script. - Run the script, and it will print the word frequencies from the input file. This script should handle the file processing and word counting as described.
Comment	2a: This refactor is extremely comprehensive—adding argument parsing, robust I/O validation, error handling with clear stderr messages, docstrings, and improved style—making it production-ready, though its verbosity might be overkill for simpler use cases.	2b: The second version succinctly tightens the original script—introducing CSV newline handling, basic float-conversion error catching, and docstrings—resulting in a lean, readable utility, albeit without flexible CLI support or full I/O/error validation.

Model Comparison and Analysis

GPT-4.1-mini's zero-shot output (1a) jumps straight to the completed script with a single, concise explanation of `clean_line`, making it immediately runnable. GPT-4.1-nano (1b) delivers the same code but wraps it in a more verbose explanation of the translation table, boosting readability for beginners.

In chain-of-thought mode, GPT-4.1 (2a) breaks down each function's purpose in neat sections—load, clean, count, main—then wires it all together, striking a balance between instruction and brevity. Codestral 25.01 (2b) likewise provides a full refactor and step-by-step commentary but leans heavier on prose and usage notes, trading some conciseness for extra context.