

Task 1: Code Summarization

Prompting Strategy 1: Zero Shot Prompt

Prompt: Summarize what the following Java method does and what it returns

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
This Java method takes a list of strings as input and returns a map that shows how many times each word appears in the list. It creates an empty HashMap, then loops through each word in the list. For each word, it either adds it to the map with a count of 1 or increments the existing count using getOrDefault. In the end, it returns the map containing the frequency of each word.	This Java method, countWordFrequency, takes a list of strings as input and calculates the frequency of each unique word within that list. It iterates through the input list, and for each word, it updates a map called freqMap. If the word is already a key in the map, its associated value (the frequency count) is incremented. If the word is encountered for the first time, it's added to the map with a frequency count of 1. Finally, the method returns the freqMap, which contains each unique word from the input list as a key and its corresponding frequency as the value.

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input: ["apple", "banana", "apple"] Output: {"apple": 2, "banana": 1}	Example 2: Input: ["cat", "dog", "cat", "cat", "dog"] Output: {"cat": 3, "dog": 2}	Example 3: Input: ["red", "blue", "red", "green", "blue", "blue"] Output: {"red": 2, "blue": 3, "green": 1}
---	---	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
The countWordFrequency method takes a list of words and returns a map showing how many times each word appears. Summary: Input: A list of strings (words). Output: A map where each key is a word and each value is the count of how often that word appears in the list. Function: It loops through the list and builds a frequency count using a HashMap. Example: Input: ["apple", "banana", "apple"] Output: {"apple": 2, "banana": 1}	As we discussed, this countWordFrequency method takes a list of strings (representing words) and calculates how many times each unique word appears in that list. It then returns a map where each unique word is a key, and its corresponding value is the count of its occurrences. The example you provided perfectly illustrates this: Input: ["apple", "banana", "apple"] Output: {"apple": 2, "banana": 1} This output shows that the word "apple" appeared twice in the input list, and the word "banana" appeared once.

Model Comparison, Comments, and Analysis: Both prompting strategies produced accurate outputs, but the few-shot approach led to more complete and polished responses. Including an example helped both models better understand the task. GPT-4 was more concise and well-structured among models, especially in the zero-shot setting, where it added an example. Gemini was more detailed but less direct. In the few-shot version, both improved, but GPT-4 still gave the clearer summary. Few-shot prompting was more effective overall, and GPT-4 consistently produced the strongest outputs.

Task 2: Bug Fixing

Prompting Strategy 1: Zero Shot Prompt

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

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>def sum_range(start, end): total = 0 for i in range(start, end + 1): total += i return total</pre>	<pre>def sum_range(start, end): total = 0 for i in range(start, end + 1): total += i return total</pre>

Prompting Strategy 2: Few-Shot Prompt

Example 1: Original: def count_to_five(): for i in range(1, 5): print(i) # Off-by-one fix: # Use range(1, 6) to include 5	Example 2: Original: def inclusive_sum(n): total = 0 for i in range(1, n): total += i return total # Off-by-one fix: # Change range(1, n) → range(1, n+1) to include	Example 3: Original: def print_range(start, end): for i in range(start, end): print(i) # Off-by-one fix: # Use range(start, end+1) to make the range inclusive
--	---	---

	n in the sum	
--	--------------	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre># Use range(start, end+1) to include the 'end' value in the sum def sum_range(start, end): total = 0 for i in range(start, end + 1): total += i return total</pre>	<pre>def sum_range(start, end): total = 0 for i in range(start, end + 1): total += i return total</pre>

Model Comparison, Comments, and Analysis: For this task, both GPT-4 and Gemini correctly identified the off-by-one error in the range function and applied the appropriate fix by adjusting the upper bound to end + 1. This was consistent across both prompting strategies. The few-shot prompt, which provided similar examples involving range-based loops, did not noticeably improve the output quality, as both models had already returned optimal solutions in the zero-shot setting. This suggests that for straightforward and well-known errors, both models perform reliably even **without** example-based prompting.

Task 3: Bug Classification

Prompting Strategy 1: Zero Shot Prompt

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

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
This is a lifetime bug (also known as a dangling pointer bug).	The bug in this C++ function is returning a pointer to a local variable. This falls under the broader category of dangling pointer issues.

Prompting Strategy 2: Few-Shot Prompt

Example 1: Code: <pre>int* createArray() { int arr[10]; return arr; }</pre> # Bug: Returning a pointer to a local array — causes undefined behavior because 'arr' is deallocated after the function returns.	Example 2: Code: <pre>char* getString() { char str[] = "hello"; return str; }</pre> # Bug: Returning address of a local array — 'str' is stored on the stack and becomes invalid after the function ends.	Example 3: Code: <pre>int* allocateMemory() { int buffer[5] = {1, 2, 3, 4, 5}; return buffer; }</pre> # Bug: Returning stack-allocated memory — this leads to undefined behavior when accessed outside the function.
---	--	---

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
Bug Type: Returning a pointer to a local variable Explanation: arr is a stack-allocated array and gets destroyed once the function exits. Returning its address creates a dangling pointer, which leads to undefined behavior if accessed later.	Bug type: Use-after-scope / dangling pointer. Local variables, such as the array arr in this function, are deallocated from the stack once the getArray function completes its execution. Returning a pointer to this deallocated memory results in a dangling pointer, and any attempt to access the memory it points to will lead to undefined behavior.

Model Comparison, Comments, and Analysis: Both GPT-4 and Gemini correctly classified it as a dangling pointer issue in both zero-shot and few-shot prompts. In the zero-shot case, GPT-4 offered a brief label, while Gemini provided a more detailed description. The few-shot strategy improved both outputs, with GPT-4 explaining stack allocation and undefined behavior, and Gemini elaborating on memory deallocation and its consequences. Both models performed well, but few-shot prompting led to more thorough and educational responses.

Task 4: Generating Email Validators

Prompting Strategy 1: Zero Shot Prompt

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

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>import re def is_valid_email(email): return bool(re.match(r'^[w\.-]+@[w\.-]+\.\w+\$', email))</pre>	<pre>import re def is_valid_email(email): pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}\$" return bool(re.fullmatch(pattern, email))</pre>

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input: "test@example.com" Output: True	Example 2: Input: "invalid-email@" Output: False	Example 3: Input: "user.name123@sub.domain.org" Output: True
--	---	---

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>import re def is_valid_email(email): return bool(re.match(r'^[\w\.-]+@[\w\.-]+\w+\$', email))</pre>	<pre>import re def is_valid_email(email): pattern = r'^[a-zA-Z0-9_%+~]@[a-zA-Z0-9-]+\.[a-zA-Z]{2,}\$' return bool(re.fullmatch(pattern, email))</pre>

Model Comparison, Comments, and Analysis: Both GPT-4 and Gemini produced accurate and practical regex solutions in both zero-shot and few-shot prompts. GPT-4 used a simple yet effective `re.match()` approach, while Gemini opted for a more robust `re.fullmatch()` with stricter pattern constraints. Interestingly, the few-shot examples did not significantly change or improve the model outputs, likely because this is a well-specified, common problem that both models already handle well. This suggests that for routine pattern-generation tasks like regex, zero-shot prompting can be just as effective as few-shot, especially when the models have already been trained on similar formats.

Task 5: Generating Flask APIs

Prompting Strategy 1: Zero Shot Prompt

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

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>from flask import Flask, jsonify app = Flask(__name__) @app.route('/greet/<username>', methods=['GET']) def greet_user(username): return jsonify({"message": f"Hello, {username}!"}) if __name__ == '__main__': app.run(debug=True)</pre>	<pre>from flask import Flask, jsonify app = Flask(__name__) @app.route('/greet/<username>') def greet(username): greeting = {"message": f"Hello, {username}! Welcome!"} return jsonify(greeting) if __name__ == '__main__': app.run(debug=True)</pre>

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input (URL): <code>/greet/Alex</code> Output (JSON): <code>{"message": "Hello, Alex!"}</code>	Example 2: Input (URL): <code>/greet/Richa</code> Output (JSON): <code>{"message": "Hello, Richa!"}</code>	Example 3: Input (URL): <code>/greet/World</code> Output (JSON): <code>{"message": "Hello, World!"}</code>
---	---	---

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>from flask import Flask, jsonify app = Flask(__name__) @app.route('/greet/<username>') def greet(username): return jsonify({"message": f"Hello, {username}!"})</pre>	<pre>from flask import Flask, jsonify app = Flask(__name__) @app.route('/greet/<username>') def greet(username): greeting = {"message": f"Hello, {username}!"} return jsonify(greeting) if __name__ == '__main__': app.run(debug=True)</pre>

Model Comparison, Comments, and Analysis: Both GPT-4 and Gemini successfully created working solutions in both zero-shot and few-shot scenarios. GPT-4 included the optional `methods=['GET']` parameter and a slightly cleaner format in zero-shot, while Gemini added a friendly "Welcome!" to the greeting. In the few-shot setup, both models changed their responses to align closely with the examples. Overall, the few-shot strategy helped reinforce expected behavior, but the models already performed well with zero-shot prompting.

Task 6: SQL Schema Design

Prompting Strategy 1: Zero Shot Prompt

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

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
-------------------------	--------------------------

<pre>-- 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 NOT NULL CHECK (rating BETWEEN 1 AND 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_review UNIQUE (user_id, book_id) -- One review per user per book);</pre>	<pre>CREATE TABLE reviews (id INTEGER PRIMARY KEY AUTOINCREMENT, user_id INTEGER NOT NULL, book_id INTEGER NOT NULL, rating INTEGER NOT NULL, review_text TEXT, -- Optional: Allow users to write a text review created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- Automatically record when the review was created updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, -- Automatically update when the review is modified FOREIGN KEY (user_id) REFERENCES users(id), FOREIGN KEY (book_id) REFERENCES books(id), -- Ensure each user can only review a book once UNIQUE (user_id, book_id), -- Add constraints for the rating value (e.g., 1 to 5 stars) CHECK (rating >= 1 AND rating <= 5)); -- Index for faster lookups on user_id CREATE INDEX idx_user_id ON reviews (user_id); -- Index for faster lookups on book_id CREATE INDEX idx_book_id ON reviews (book_id); -- Index for faster lookups on user_id and book_id together (for checking uniqueness) CREATE UNIQUE INDEX idx_user_book ON reviews (user_id, book_id);</pre>
--	--

Prompting Strategy 2: Few-Shot Prompt

<p>Example 1: Input: A user table with id and name, a book table with id and title, and a review table with id, user_id, book_id, and rating. Output: CREATE TABLE users (id INT PRIMARY KEY, name VARCHAR(100)); CREATE TABLE books (id INT PRIMARY KEY, title VARCHAR(255)); CREATE TABLE reviews (id INT PRIMARY KEY, user_id INT, book_id INT, rating INT, FOREIGN KEY (user_id) REFERENCES users(id), FOREIGN KEY (book_id) REFERENCES books(id));</p>	<p>Example 2: Input: Tables with appropriate primary keys and foreign keys to link users and books through reviews. Output: CREATE TABLE users (id SERIAL PRIMARY KEY, name TEXT NOT NULL); CREATE TABLE books (id SERIAL PRIMARY KEY, title TEXT NOT NULL); CREATE TABLE reviews (id SERIAL PRIMARY KEY, user_id INT NOT NULL, book_id INT NOT NULL, rating INT CHECK (rating BETWEEN 1 AND 5), FOREIGN KEY (user_id) REFERENCES users(id), FOREIGN KEY (book_id) REFERENCES books(id));</p>	<p>Example 3: Input: Design a normalized SQL schema for a book review app. Output: CREATE TABLE users (id INT PRIMARY KEY, name VARCHAR(255)); CREATE TABLE books (id INT PRIMARY KEY, title VARCHAR(255)); CREATE TABLE reviews (id INT PRIMARY KEY, user_id INT, book_id INT, rating INT, FOREIGN KEY (user_id) REFERENCES users(id), FOREIGN KEY (book_id) REFERENCES books(id), UNIQUE (user_id, book_id));</p>
--	--	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>CREATE TABLE users (id SERIAL PRIMARY KEY, name VARCHAR(100) NOT NULL); CREATE TABLE books (id SERIAL PRIMARY KEY, title VARCHAR(255) NOT NULL); CREATE TABLE reviews (id SERIAL PRIMARY KEY, user_id INT NOT NULL, book_id INT NOT NULL, rating INT CHECK (rating BETWEEN 1 AND 5), FOREIGN KEY (user_id) REFERENCES users(id), FOREIGN KEY (book_id) REFERENCES books(id), UNIQUE (user_id, book_id) -- Ensures one review per user-book pair);</pre>	<pre>CREATE TABLE users (id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(255) NOT NULL); CREATE TABLE books (id INT PRIMARY KEY AUTO_INCREMENT, title VARCHAR(255) NOT NULL); CREATE TABLE reviews (id INT PRIMARY KEY AUTO_INCREMENT, user_id INT NOT NULL, book_id INT NOT NULL, rating INT NOT NULL CHECK (rating BETWEEN 1 AND 5), FOREIGN KEY (user_id) REFERENCES users(id), FOREIGN KEY (book_id) REFERENCES books(id), UNIQUE (user_id, book_id));</pre>

Model Comparison, Comments, and Analysis: The zero-shot responses from both GPT-4 and Gemini were well-structured. GPT-4 included thoughtful constraints like ON DELETE CASCADE, while Gemini added timestamps, indexes, and a review_text field for enhanced utility. In contrast, the few-shot responses focused more on producing minimal, clean schemas that still met all functional requirements but omitted some of the

additional features. Overall, both models performed well in both strategies, but the zero-shot approach outputted more comprehensive schemas, especially for Gemini. This suggests that for declarative languages like SQL, zero-shot prompting may be sufficient when **clear intent** is provided.

Task 7: Null Dereference Detection

Prompting Strategy 1: Zero Shot Prompt

Prompt: Identify any null dereference risk:

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>public int getLength(String s) { if (s == null) { throw new IllegalArgumentException("Input string cannot be null"); } return s.length(); }</pre>	<pre>import java.util.Optional; public int getLength(String s) { return Optional.ofNullable(s) .map(String::length) .orElse(0); // Or orElseThrow(IllegalArgumentException::new)}</pre>

Prompting Strategy 2: Few-Shot Prompt

Example 1: Code: <pre>public int getLength(String s) { return s.length(); }</pre> Risk: If s is null, calling s.length() will throw a NullPointerException.	Example 2: Code: <pre>public boolean isEmpty(String s) { return s.isEmpty(); }</pre> Risk: If s is null, s.isEmpty() will crash the program with a NullPointerException.	Example 3: Code: <pre>public char firstChar(String input) { return input.charAt(0); }</pre> Risk: If input is null, the call to charAt(0) will throw a NullPointerException.
---	--	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>Code: public int getLength(String s) { return s.length(); // What if s is null? }</pre> Risk: If s is null, calling s.length() will throw a NullPointerException at runtime. This occurs because you're trying to call a method on a null reference, which Java doesn't allow. To avoid this, you should check if s is null before calling .length().	<pre>Code: public int getLength(String s) { return s.length(); // What if s is null? }</pre> Risk: If s is null, attempting to call the length() method on it (s.length()) will result in a NullPointerException. This is because null is not an object, and you cannot access members or invoke methods on a null reference.

Model Comparison, Comments, and Analysis: In the zero-shot, GPT-4 explicitly threw an IllegalArgumentException if the input is null, while Gemini provided a more modern approach using Optional.ofNullable, returning a default value when the string is null. In the few-shot case, both models clearly articulated the risk of invoking length () on a null reference and explained why it leads to a NullPointerException. Overall, while both models were consistent across strategies, the zero-shot responses were slightly more implementation-focused, whereas the few-shot responses provided clearer conceptual reasoning. This makes few-shot more useful for educational explanations, while zero-shot is suitable when quick solutions are preferred.

Task 8: CSV Parser Variants

Prompting Strategy 1: Zero Shot Prompt

Prompt: Improve the parser to support quoted fields

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>import csv from io import StringIO def parse_csv_line(line): return next(csv.reader(StringIO(line)))</pre>	<pre>import re def parse_csv_line_regex(line): pattern = r'(?:"(["]*)") ([^\s,]+)' matches = re.findall(pattern, line) result = [group[0] if group[0] else group[1] for group in matches] return result</pre>

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input: "name,age,city" Output: ["name", "age", "city"]	Example 2: Input: "John,25,\"New York, NY\"" Output: ["John", "25", "New York, NY"]	Example 3: Input: "\"Doe, Jane\",30,\"Los Angeles, CA\"" Output: ["Doe, Jane", "30", "Los Angeles, CA"]
--	--	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
-------------------------	--------------------------

<pre>import csv from io import StringIO def parse_csv_line(line): return next(csv.reader(StringIO(line)))</pre>	<pre>import csv import io def parse_csv_line(line): f = io.StringIO(line) reader = csv.reader(f) for row in reader: return row return []</pre>
---	--

Model Comparison, Comments, and Analysis: In the zero-shot setup, GPT-4 used the csv.reader module wrapped with StringIO, while Gemini opted for a regex-based approach. Gemini's solution is smart, but it might not deal with unusual situations as well as Python's built-in CSV module. In the few-shot setup, both models converged on the more reliable csv.reader strategy, showing improved alignment with best practices for parsing quoted fields. Overall, few-shot prompting led to more standardized/safer implementations in both models.

Task 9: Data Class to API Conversion

Prompting Strategy 1: Zero Shot Prompt

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

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>import io.ktor.application.* import io.ktor.http.* import io.ktor.response.* import io.ktor.serialization.* @Serializable data class Product(val id: Int, val name: String, val price: Double) val products = mutableListOf<Product>() fun main() { embeddedServer(Netty, port = 8080) { install(ContentNegotiation) { json() } routing { // GET all products get("/products") { call.respond(products) } // POST a new product post("/products") { val product = call.receive<Product>() products.add(product) call.respond(HttpStatusCode.Created, product) } } }.start(wait = true)}</pre>	<pre>import io.ktor.server.application.* import io.ktor.http.* @Serializable data class Product(val id: Int, val name: String, val price: Double) // In-memory storage for products val products = mutableListOf<Product>() var nextProductId = 1 fun main(args: Array<String>) { io.ktor.server.netty.EngineMain.main(args) }fun Application.module() { install(ContentNegotiation) { json() }routing { // GET all products get("/products") { if (products.isEmpty()) { call.respond(HttpStatusCode.NoContent, "No products available") } else { call.respond(HttpStatusCode.NotFound, "Product not found") } } // POST a new product post("/products") { val product = call.receive<Product>() products.add(newProduct) call.respond(HttpStatusCode.Created, newProduct) // Respond with 201 Created and the new product}}</pre>

Prompting Strategy 2: Few-Shot Prompt

<p>Example 1: Input: A data class in Kotlin: data class Book(val id: Int, val title: String) Output: @get("/book/{id}") fun getBook(@Path("id") id: Int): Book { return Book(id, "Sample Title") }</p>	<p>Example 2: Input: data class User(val id: Int, val name: String) Output: fun Application.module() { routing { get("/user/{id}") { val id = call.parameters["id"]?.toInt() ?: return@get }call.respond(HttpStatusCode.BadRequest) call.respond(User(id, "Alice")) }post("/user") { val user = call.receive<User>() call.respond(HttpStatusCode.Created, user)}}}</p>	<p>Example 3: Input: data class Product(val id: Int, val name: String, val price: Double) Output: fun Application.module() { routing { get("/product/{id}") { val id = call.parameters["id"]?.toInt() ?: return@get }call.respond(HttpStatusCode.BadRequest) call.respond(Product(id, "Example", 19.99))}post("/product") { val product = call.receive<Product>() call.respond(HttpStatusCode.Created, product)}}}</p>
---	---	---

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
-------------------------	--------------------------

<pre> import io.ktor.application.* import io.ktor.http.* import io.ktor.response.* import io.ktor.request.* import io.ktor.routing.* import io.ktor.server.engine.* import io.ktor.server.netty.* import io.ktor.features.* import io.ktor.serialization.* import kotlinx.serialization.Serializable @Serializable data class Product(val id: Int, val name: String, val price: Double) fun Application.module() { install(ContentNegotiation) { json() } routing { get("/product/{id}") { val id = call.parameters["id"]?.toIntOrNull() ?: return@get call.respond(HttpStatusCode.BadRequest, "Missing or invalid ID") } call.respond(Product(id, "Sample Product", 29.99)) } post("/product") { val product = call.receive<Product>() call.respond(HttpStatusCode.Created, product)}} </pre>	<pre> import io.ktor.server.application.* import io.ktor.server.response.* import io.ktor.server.routing.* import io.ktor.server.netty.* import io.ktor.serialization.kotlinx.json.* import io.ktor.server.plugins.contentnegotiation.* import io.ktor.server.request.* import io.ktor.http.HttpStatusCode import kotlinx.serialization.Serializable @Serializable data class Product(val id: Int, val name: String, val price: Double) fun Application.module() { routing { get("/product/{id}") { val id = call.parameters["id"]?.toInt() ?: return@get call.respond(HttpStatusCode.BadRequest) call.respond(Product(id, "Example Product", 19.99)) //Hardcoded, as in the examples } post("/product") { val product = call.receive<Product>() call.respond(HttpStatusCode.Created, product)}} fun main(args: Array<String>) { EngineMain.main(args) } </pre>
--	---

Model Comparison, Comments, and Analysis: Both models returned functional Ktor APIs in zero-shot and few-shot settings. Gemini's zero-shot output was more detailed, including validation and multiple endpoints, while GPT-4's was simpler. In the few-shot, both models produced shorter, more uniform outputs that reflected the prompt examples, though they leaned on hardcoded responses. Overall, zero-shot prompted more complex behavior, while few-shot led to more controlled, efficient, and minimal code.

Task 10: Function Summarization

Prompting Strategy 1: Zero Shot Prompt

Prompt: Write a brief summary of the function

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
The reverse_words function takes a sentence as input and returns a new sentence with the order of words reversed. It splits the sentence into words, reverses the list of words, and joins them back into a single string with spaces.	The reverse_words function takes a string (sentence) as input and returns a new string with the order of the words reversed. It achieves this by: 1. Splitting the input sentence into a list of individual words using the split() method. 2. Reversing the order of words in the list using slicing [::-1]. 3. Joining the reversed list of words back into a single string, separated by spaces, using the join() method.

Prompting Strategy 2: Few-Shot Prompt

Example 1: Function: def square(n): return n * n Summary: This function returns the square of a given number.	Example 2: Function: def get_first_item(lst): return lst[0] Summary: This function retrieves the first element from a list.	Example 3: Function: def to_uppercase(s): return s.upper() Summary: This function converts a string to uppercase letters.
Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:	
Function: def reverse_words(sentence): return ' '.join(sentence.split()[::-1])	Function: def reverse_words(sentence): return ' '.join(sentence.split()[::-1])	

Summary: This function returns the input sentence with the words in reverse order	Summary: This function reverses the order of the words in a given sentence.
---	---

Model Comparison, Comments, and Analysis:Both GPT-4 and Gemini created very similar summaries that matched the prompt well. The models understood how to change sentences and explained this clearly, whether they had seen examples before or not. Their outputs showed high lexical similarity, with only slight changes in wording, indicating a strong overlap when measured by BLEU or ROUGE scores. Since the task was simple, there wasn't much room for variation, so using a few examples didn't noticeably improve the results. Nevertheless, both models followed the structure and intent of the prompt effectively, showing consistency and dependability in simpler language tasks.

Task 11: Prompt from Code Comments

Prompting Strategy 1: Zero Shot Prompt

Prompt: Write a prompt that could generate the code:

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
Prompt: Write a Python function that checks whether a given number is a prime number. The function should return True if the number is prime and False otherwise. Use an efficient approach that only checks divisibility up to the square root of the number.	Write a Python function that checks if a number is prime.

Prompting Strategy 2: Few-Shot Prompt

Example 1: Comment: # This function calculates the factorial of a number Code Output Expected: def factorial(n): if n == 0: return 1 return n * factorial(n - 1) Prompt: Generate Python code that implements the following behavior: "This function calculates the factorial of a number."	Example 2: Comment: # This function returns True if the number is even Code Output Expected: def is_even(n): return n % 2 == 0 Prompt: Create a Python function using the comment: "This function returns True if the number is even."
--	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
Prompt: Create a Python function using the comment: "This function checks if a number is prime." The function should return True if the number is prime and False otherwise, using an efficient method that checks divisibility up to the square root of the number.	Prompt: Generate a Python function that checks if a number is prime. The function should be named is_prime, take an integer n as input, and return True if n is prime, and False otherwise. Include a comment at the beginning of the function explaining its purpose. The code should implement an optimized primality check by iterating only up to the square root of n

Model Comparison, Comments, and Analysis: In the zero-shot strategy, GPT-4 generated a more descriptive and structured prompt, specifying that the function should return a boolean and use an efficient method involving the square root of the number. Gemini's output, while functionally accurate, was more concise and less directive. In the few-shot setting, both models produced more quality and precise prompts that reflected the goal of generating a prime-checking function. GPT-4 included the expected logic and constraints, while Gemini expanded on implementation details like naming conventions and added comments. Although both were successful, the few-shot approach enhanced the completeness and clarity of the prompt, especially in guiding the model toward optimized code. In this case, few-shot prompting demonstrated better control over prompt structure and intent articulation across both models.

Task 12: Fixing Factorial Bug

Prompting Strategy 1: Zero Shot Prompt

Prompt: Fix the bug when input is 0:

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
def factorial(n): result = 1 for i in range(1, n + 1): result *= i return result	def factorial(n): if n == 0: return 1 # Base case: 0! is 1 result = 1 for i in range(1, n + 1): # Iterate up to and including n result *= i return result

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input: factorial(0) Expected Output: 1 Prompt: The following Python function incorrectly calculates the factorial for 0. Fix the function so that it correctly handles input 0 and returns 1.	Example 2: Input: 0 Output: 1 Prompt: Update the factorial function to handle the edge case when n == 0. The factorial of 0 is defined as 1.
--	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>def factorial(n): result = 1 for i in range(1, n + 1): result *= i return result</pre>	Prompt: Modify the given factorial function to ensure it correctly returns 1 when the input is 0.

Model Comparison, Comments, and Analysis: Both GPT-4 and Gemini successfully corrected the factorial function in the zero-shot setting by either modifying the loop bounds or explicitly adding a base case for $n == 0$. However, in the few-shot setting, GPT-4 repeated its correct solution, while Gemini misinterpreted the prompt and returned a description of the fix instead of actual code. This indicates a failure to follow the output format expected in few-shot prompting. We could attribute this to improper examples, however, GPT-4 was able to follow instructions using context. While zero-shot responses may score well in similarity measures like CodeBLEU or AST-based comparisons, the few-shot output from Gemini will likely fall short in those same measures because it does not provide a code response.

Task 13: Linked List Node Deletion

Prompting Strategy 1: Zero Shot Prompt

Prompt: Implement node deletion by value:

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>#include <stdlib.h> struct Node { int data; struct Node* next; }; void deleteNode(struct Node** head, int key) { struct Node* temp = *head; struct Node* prev = NULL; // If head node itself holds the key if (temp != NULL && temp->data == key) { *head = temp->next; free(temp); return;} // Search for the key to be deleted 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 and free memory prev->next = temp->next; free(temp);}</pre>	<pre>#include <stdio.h> #include <stdlib.h> struct Node { int data; struct Node *next;}; void deleteNode(struct Node **head, int key) { struct Node *current = *head; struct Node *prev = NULL; // Handle the case where the list is empty if (*head == NULL) { return; // Nothing to delete } (CODE OMITTED - FULL CODE IN RAW FILES) // Delete the last remaining node deleteNode(&head, 20); printf("List after deleting 20: "); printList(head); // Check deleting from an empty list deleteNode(&head, 20); printf("List after deleting from empty list: "); printList(head); return 0;}</pre>

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input: Head = [1 → 2 → 3], Key = 2 Output: [1 → 3]	Example 2: Input: Head = [5 → 6 → 7 → 8], Key = 5 Output: [6 → 7 → 8]	Example 3: Input: Head = [10 → 20 → 30], Key = 30 Output: [10 → 20]
--	--	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>struct Node { int data; struct Node* next; }; void deleteNode(struct Node** head, int key) { struct Node* temp = *head, *prev = NULL; while (temp && temp->data != key) { prev = temp; temp = temp->next; } if (!temp) return;</pre>	<pre>#include <stdlib.h> struct Node { int data; struct Node* next; }; void deleteNode(struct Node** head, int key) { struct Node *curr = *head, *prev = NULL; if (!curr) return; // Empty list if (curr->data == key) { *head = curr->next; free(curr); return; }</pre>

<pre> if (!prev) *head = temp->next; else prev->next = temp->next; free(temp); } </pre>	<pre> while (curr && curr->data != key) { prev = curr; curr = curr->next;} if (!curr) return; // Key not found prev->next = curr->next; free(curr); } </pre>
--	--

Model Comparison, Comments, and Analysis: Both GPT-4 and Gemini successfully identified/implemented the correct logic for deleting a node by value in a singly linked list, handling edge cases such as deletion at the head and key not found. The zero-shot outputs from both models showed strong baseline understanding, though Gemini's response included extra test code and formatting inconsistencies that made it harder to isolate the implementation logic. In contrast, few-shot prompting encouraged both models to produce **more concise and focused solutions**. GPT-4's few-shot output cleanly handled all cases in fewer lines, with Gemini also improving clarity compared to its zero-shot variant. This task shows that both models can produce accurate results, but using examples leads to more consistent and organized outputs, which is better for tasks like managing linked lists.

Task 14: Recursive Function Completion

Prompting Strategy 1: Zero Shot Prompt

Prompt: Complete the recursive function for Fibonacci:

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre> def fibonacci(n): if n <= 1: return n return fibonacci(n - 1) + fibonacci(n - 2) </pre>	<pre> def fibonacci(n): if n <= 1: # Base cases: F(0) = 0, F(1) = 1 return n else: return fibonacci(n - 1) + fibonacci(n - 2) </pre>

Prompting Strategy 2: Chain-of-Thought Prompt

<p>Complete the recursive function to return the nth Fibonacci number. Before writing code, think through the logic step-by-step. To solve this, we need to return the nth number in the Fibonacci sequence. The Fibonacci sequence is defined as: fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2) for n >= 2 So we need to:</p> <ol style="list-style-type: none"> 1. Handle the base cases: if n == 0, return 0; if n == 1, return 1. 2. For other cases, recursively call fibonacci(n-1) and fibonacci(n-2) and return their sum.

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre> def fibonacci(n): if n == 0: return 0 if n == 1: return 1 return fibonacci(n - 1) + fibonacci(n - 2) </pre>	<pre> def fibonacci(n): if n == 0: return 0 elif n == 1: return 1 else: return fibonacci(n - 1) + fibonacci(n - 2) </pre>

Model Comparison, Comments, and Analysis: Both GPT-4 and Gemini successfully created the recursive Fibonacci function. They did this in both the zero-shot and chain-of-thought settings. The results were the same in structure, including the right base case handling and correct recursive calls. CoT prompting breaks down recursion logic into clear steps, which can make the model more reliable for complex tasks. Although the output improvement isn't significant, CoT still helps by making the reasoning process easy to understand. This transparency is beneficial for debugging or adjusting the logic for more advanced recursive problems.

Task 15: Constructor Completion

Prompting Strategy 1: Zero Shot Prompt

Prompt: Complete the class constructor

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre> class Person: def __init__(self, name, age, email=None): self.name = name self.age = age self.email = email </pre>	<pre> class Person: def __init__(self, name, age, email=None): self.name = name self.age = age self.email = email </pre>

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input: class Car: def __init__(self, make, model, year): # TODO: Add constructor pass Output: class Car: def __init__(self, make, model, year): self.make = make self.model = model self.year = year	Example 2: Input: class Student: def __init__(self, name, age, gpa=None): # TODO: Add constructor pass Output: class Student: def __init__(self, name, age, gpa=None): self.name = name self.age = age self.gpa = gpa
---	---

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
class Person: def __init__(self, name, age, email=None): self.name = name self.age = age self.email = email	class Person: def __init__(self, name, age, email=None): self.name = name self.age = age self.email = email

Model Comparison, Comments, and Analysis: In both the zero-shot and few-shot settings, GPT-4 and Gemini correctly completed the constructor by adding name, age, and an optional email field as parameters, and initializing them as instance variables. The syntax and logic were accurate in all cases, demonstrating that this task is straightforward enough to be handled well without additional examples. While the few-shot prompt reinforces the pattern, it doesn't significantly enhance output quality for this task compared to the zero-shot setup. This task reflects minimal variance between strategies or models, suggesting high prompt-responsiveness due to the simplicity and clarity of the original instruction.

Task 16: Binary Search Completion

Prompting Strategy 1: Zero Shot Prompt

Prompt: Complete the binary search implementation

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
public int binarySearch(int[] arr, int target) { int left = 0, right = arr.length - 1; while (left <= right) { int mid = (left + right) / 2; if (arr[mid] == target) return mid; else if (arr[mid] < target) left = mid + 1; else right = mid - 1; } return -1; }	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; // Target found at mid } else if (arr[mid] < target) { left = mid + 1; // Target is in the right half } else { right = mid - 1; // Target is in the left half } } return -1; // Target not found }

Prompting Strategy 2: Few-Shot Prompt

Example 1: Input (Incomplete Code): public int binarySearch(int[] arr, int target) { int left = 0, right = arr.length - 1; while (left <= right) { int mid = (left + right) / 2; // TODO: Fill in logic } return -1; } Output: if (arr[mid] == target) return mid; else if (arr[mid] < target) left = mid + 1; else right = mid - 1;	Example 2: Input (Incomplete Code): public int find(int[] nums, int x) { int l = 0, r = nums.length - 1; while (l <= r) { int m = (l + r) / 2; // Fill here } return -1; } Output: if (nums[m] == x) return m; else if (nums[m] < x) l = m + 1; else r = m - 1;
---	--

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
-------------------------	--------------------------

<pre>public int binarySearch(int[] arr, int target) { int left = 0, right = arr.length - 1; while (left <= right) { int mid = (left + right) / 2; if (arr[mid] == target) return mid; else if (arr[mid] < target) left = mid + 1; else right = mid - 1; } return -1; }</pre>	<p>I've filled in the // TODO section with the conditional logic to compare the middle element with the target and adjust the search bounds, just like in your provided examples.</p> <pre>if (arr[mid] == target) return mid; else if (arr[mid] < target) left = mid + 1; else right = mid - 1;</pre>
--	---

Model Comparison, Comments, and Analysis: In the binary search task, both GPT-4 and Gemini generated correct outputs using zero-shot and few-shot strategies. The underlying logic remained consistent across both models, involving comparisons of the middle element and adjusting the search bounds accordingly. GPT-4's few-shot output was particularly clean and concise, while Gemini's output was accurate but tended to focus on explaining its reasoning. Overall, the few-shot approach slightly improved formatting, but both models performed equally well in terms of functionality.

Task 17: Self-Consistency Bug Fixing

Prompting Strategy 1: Zero Shot Prompt

Prompt: Resolve inconsistency between function name and logic:

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>1. bool isEven(int x) { return x % 2 == 0; } 2. bool isOdd(int x) { return x % 2 != 0; }</pre>	<pre>// Returns true if x is even bool isEven(int x) { return x % 2 == 0; }</pre>

Prompting Strategy 2: Chain-of-Thought Prompt

<p>The function below is named isOdd, suggesting it should return true when the input is odd. However, the objective/comment says it must be even. First, determine whether the logic matches the name. Then, correct the logic if needed.</p> <p>// Supposed to return true if x is even</p> <pre>bool isOdd(int x) { return x % 2 == 0; // Logic contradicts function name}</pre>

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>bool isEven(int x) { return x % 2 == 0; }</pre>	<p>Since the comment describes the intended behavior, I'll correct the logic to match the comment and rename the function. Here's the corrected code: // Returns true if x is even</p> <pre>bool isEven(int x) { return x % 2 == 0; }</pre>

Model Comparison, Comments, and Analysis: In the zero-shot setting, both models directly inferred that the logic contradicted the function name and returned corrected versions, either by renaming the function (isEven) or adjusting the return condition to reflect the intended behavior. After using COT prompting, the models identified that the objective was that the method was supposed to check if it was even, and both adjusted the code accordingly. Gemini even made a statement reflecting its updated logic (saying that it is taking the comment into account). Though the first zero-shot was correct, the COT prompt offered minor tweaks and more specificity (that might be impactful in higher-level code).

Task 18: Prompt Chaining: Bug Identification → Fix

Prompting Strategy 1: Zero Shot Prompt

Prompt: Identify and fix the bug in the following code

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>function isEven(n) { return n % 2 === 0; }</pre>	<pre>function isEven(n) { return n % 2 === 0; // Corrected to return true or false }</pre>

Prompting Strategy 2: Prompt Chaining

Step 1: Bug Identification	Step 2: Bug Fix
----------------------------	-----------------

Prompt: What's wrong with this function? function isEven(n) { return n % 2; // Returns 1 or 0}	Prompt: Fix the function so it correctly returns a boolean value. Expected Output: function isEven(n) { return n % 2 === 0;}
--	---

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
Issue: The function returns 1 for odd numbers and 0 for even numbers, not actual boolean values true or false Fixed Code: function isEven(n) { return n % 2 === 0;}	function isEven(n) { return n % 2 === 0; }

Model Comparison, Comments, and Analysis: Both GPT-4 and Gemini successfully identified and resolved the logical issue in the isEven function across prompting strategies. Under zero-shot prompting, each model directly returned the corrected version of the function, replacing the numeric return value (n % 2) with a proper boolean comparison (n % 2 === 0). In prompt chaining, GPT-4 first explained that the original function returned 1 or 0 instead of true or false, and then followed up with the corrected implementation. While both strategies were effective, prompt chaining offered a clearer breakdown of the error and correction process, making it more interpretable and useful for debugging or educational contexts.

Task 19: Summary Decomposition

Prompting Strategy 1: Zero Shot Prompt

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

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
Original Comment: // Function that validates an input, calculates the square, and returns the result Decomposed Steps: Validate the input: Check if x is a negative number. If it is, return -1 to indicate invalid input. Calculate the square: If the input is valid (i.e., non-negative), compute the square of x. Return the result: Return the squared value as the result.	High-level comment/summary: // Function that validates an input, calculates square, and returns result Decomposed logical steps: Validate input: Check if the input x is less than 0. Handle invalid input: If x is less than 0, return -1 to indicate an error. Calculate square: If x is not less than 0, calculate the square of x (i.e., x * x). Return result: Return the calculated square of x.

Prompting Strategy 2: Few-Shot Prompt

You are a senior C++ software engineer mentoring a junior developer. They've written the following function, and you've left a comment describing what it does. Break the comment down into three logical steps that explain what the function actually does at the code level. Use clear, technical phrasing that will help the junior developer understand both the logic and intent behind the code.

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
So this function first checks if the input x is negative. If it is, it immediately returns -1—this is a simple form of input validation to make sure we're only working with non-negative numbers. If the input is valid (zero or positive), the function then calculates the square of x by multiplying it by itself, and returns that result. Basically, it's a two-part check: first we make sure the input is okay, and then we do the actual work (squaring the number) and return it. The -1 acts as a signal that something was wrong with the input.	Okay, let's break down what this function, process, actually does: <ul style="list-style-type: none">Input Validation: The function first performs input validation by checking if the integer argument x is less than zero.Error Handling: If x is negative, the function returns -1. This return value signals an error condition, indicating that the input was invalid according to the function's logic.Square Calculation and Return: If x is not negative (i.e., it's zero or positive), the function calculates the square of x by multiplying x by itself (x * x). This calculated square is then returned as the result of the function.

Model Comparison, Comments, and Analysis: In the zero-shot strategy, both GPT-4 and Gemini provided clear breakdowns of the high-level comment. They effectively divided the logic into three stages: input validation, computation, and return. Their outputs were similar in both structure and meaning, making them strong candidates for metrics like BLEU or ROUGE. In the role-playing strategy, both models used a conversational tone, showing how a human might explain the function simply. GPT-4's version was smoother and more friendly for teaching, while Gemini was more technical but still accurate. This approach helped make the information clearer for non-technical audiences and showed that both models can adjust their style and tone based on the prompt. Overall, their close alignment across models and strategies suggests that this task could achieve high similarity scores with methods like cosine similarity using Sentence-BERT.

Task 20: Purpose Inference → Completion

Prompting Strategy 1: Zero Shot Prompt

Prompt: Complete the function based on intent

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<pre>def calculate_average(scores): total = sum(scores) return total / len(scores) if scores else 0</pre> <p>This version calculates the total using sum(), then returns the average. It also safely handles empty lists by returning 0 if scores is empty.</p>	<pre>def calculate_average(scores): total = 0 if not scores: return 0 # Handle the case of an empty list for score in scores: total += score return total / len(scores)</pre>

Prompting Strategy 2: Self-Consistency Prompting

<p>Task: Complete the following Python function to calculate the average of a list of scores.</p> <p>Prompt Template: Why does this code not compute the correct average? Collect each answer (e.g., 3 total), extract their conclusions, and return the most common one.</p>

Model 1 (GPT-4) Output:	Model 2 (Gemini) Output:
<ol style="list-style-type: none">1. The function initializes total to 0 but never adds the elements of scores to it, so the sum remains 0. It also doesn't return anything, so the function always returns None.2. It's missing the loop or built-in sum() to calculate the total score, and it doesn't divide by the number of elements to get the average.3. There's no logic to compute or return the average. It just has a pass, which means it does nothing. It should be using sum(scores) / len(scores). <p>Fixed code: def calculate_average(scores): total = sum(scores) return total / len(scores) if scores else 0</p>	<p>The code initializes total to 0, but it never actually adds the scores from the scores list to this total. Also, it lacks the final step of dividing the total by the number of scores to calculate the average.</p>

Model Comparison, Comments, and Analysis: In zero shot, GPT-4 used a simple and clear approach with sum() and a fallback for empty lists, demonstrating robustness. Gemini used a loop for summation, which is less elegant but still correct. GPT-4 effectively identified issues in the original code, suggested a fix, and reached the correct solution using sum() and a safe division. Gemini, on the other hand, did not actually complete the self-consistency flow. It pointed out that the code fails to sum the values and lacks the division step, but it never presented a fixed version of the code. That breaks the self-consistency format entirely, which is supposed to gather multiple completions and return the most consistent solution. Gemini offered analysis but failed to follow through with a valid corrected output. GPT-4 was more aligned with the self-consistency methodology, diagnosing, fixing, and confirming the solution. Gemini's response was incomplete and doesn't satisfy the expectations of this prompting strategy. For this task, GPT-4 clearly outperformed Gemini.