GenAI for Software Development: Assignment 3

Benjamin Normann
benormann@wm.edu

Introduction

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

Task 1: Code Summarization (Java)

Prompting Strategy 1: Zero-shot Prompt:

```
Summarize the functionality of the following method:

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

GPT-4o Response:

```
The method `countWordFrequency` takes a list of strings (`words`) as input and returns a map (`Map<String, Integer>`) that represents the frequency of each word in the list. Here's how it works:
1. It initializes an empty `HashMap` called `freqMap` to store words as keys and their respective frequencies as values.
2. It iterates through each word in the input list (`words`).
3. For each word, it checks if the word already exists in the map:
  - If it exists, it increments the count (value) by 1.
  - If it does not exist, it adds the word to the map with an initial count of 1.
4. Finally, it returns the `freqMap` containing the word-frequency pairs.
In summary, the method calculates and returns a frequency distribution of words from the input list.
```

Response time: 3.85 seconds

Llama-3.1 Response:

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

Here's a step-by-step breakdown:

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

Example usage:
```java
List<String> words = Arrays.asList("apple", "banana", "apple", "orange", "banana", "banana");
Map<String, Integer> freqMap = countWordFrequency(words);
// freqMap would contain: {"apple": 2, "banana": 3, "orange": 1}
```
Response time: 7.58 seconds

Few-shot Prompt:

Summarize the functionality of the following method:
```
public Map<String, Integer> countWordFrequency(List<String> words) {
        Map<String, Integer> freqMap = new HashMap<>();
        for (String word : words) {
                freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);
        }
        return freqMap;
}
```
Example 1:
This method counts the frequency of each word in a list and returns a map where keys are words and values are their occurrence counts.
Example 2:
This method creates a frequency map of words. It iterates through a list of strings, incrementing the count for each word in a HashMap, and returns the resulting frequency map.

GPT-4o Response:

The method `countWordFrequency` takes a list of strings (`words`) as input and returns a map where each unique word is a key, and its corresponding value is the number of times that word appears in the list. It works as follows:
1. Initializes an empty `HashMap` (`freqMap`) to store word frequencies.
2. Iterates through each word in the input list.
3. For each word, it updates the frequency in the map by using `getOrDefault` to retrieve the current count (defaulting to 0 if the word is not already in the map), then increments it by 1.
4. After processing all words, it returns the `freqMap` containing the word-frequency pairs.
Key Points:
- Handles duplicate words by incrementing their count in the map.
- Uses `getOrDefault` to simplify handling of new words.
- Returns a `HashMap` with word-frequency mappings.

```
### Example Input and Output:
**Input:** `["apple", "banana", "apple", "orange", "banana", "banana"]`
**Output:** `{"apple"=2, "banana"=3, "orange"=1}`
```

Response time: 4.08 seconds

Llama-3.1 Response:

```
The provided method, `countWordFrequency`, takes a list of words as input and returns a map where each word is a key, and its corresponding value is the frequency of that word in the list. It achieves this by iterating over each word in the list and incrementing its count in the map. If the word is not already in the map, it defaults to a count of 0 before incrementing.
Both example descriptions accurately summarize the functionality of the method. The first example focuses on the purpose of the method, while the second example provides a more detailed explanation of how it works.
Here's a more concise version of the descriptions:
"This method returns a map of word frequencies from a given list of words."
or
"This method creates a frequency map of words by iterating over a list and counting each word's occurrence."
```

Response time: 5.97 seconds

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

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

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

Prompt & Responses given in lines 111-290 of *model_comparison_results.txt*

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

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

Task 3: Bug Classification (C++)

Prompt & Responses given in lines 292-487 of *model_comparison_results.txt*

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

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

Task 4: Generating Email Validators (Python + Regex)

Prompt & Responses given in lines 489-684 of *model_comparison_results.txt*

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

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

Task 5: Generating Flask APIs (Python)

Prompt & Responses given in lines 686-894 of *model_comparison_results.txt*

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

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

Task 6: SQL Schema Design (SQL)

Prompt & Responses given in lines 896-1217 of model_comparison_results.txt

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

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

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

Task 7: Null Dereference Detection (Java)

Prompt & Responses given in lines 1219-1486 of model_comparison_results.txt

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

Comparison: This a tossup as to which to use. They are both accurate and complete.

Task 8: CSV Parser Variants (Python)

Prompt & Responses given in lines 1488-1775 of model_comparison_results.txt

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

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

Task 9: Data Class to API Conversion (Kotlin)

Prompt & Responses given in lines 1777-2289 of model_comparison_results.txt

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

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

Task 10: Function Summarization (Python)

Prompt & Responses given in lines 2294-2375 of model_comparison_results.txt

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

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

Task 11: Prompt from Code Comments (Python)

Prompt & Responses given in lines 2376-2454 of model_comparison_results.txt

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

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

Task 12: Fixing Factorial Bug (Python)

Prompt & Responses given in lines 2455-2666 of model_comparison_results.txt

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

Comparison: Select Llama for robust code.

Task 13: Linked List Node Deletion (C)

Prompt & Responses given in lines 2667-3210 of model_comparison_results.txt

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

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

Task 14: Recursive Function Completion (Python)

Prompt & Responses given in lines 3211-3531 of model_comparison_results.txt

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

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

Task 15: Constructor Completion (Python)

Prompt & Responses given in lines 3532-3753 of model_comparison_results.txt

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

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

Task 16: Binary Search Completion (Java)

Prompt & Responses given in lines 3754-4070 of model_comparison_results.txt

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

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

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

Prompt & Responses given in lines 4071-4258 of model_comparison_results.txt

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

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

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

Prompt & Responses given in lines 4259-4431 of model_comparison_results.txt

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

Comparison: No appreciable difference.

Task 19: Summary Decomposition (C++)

Prompt & Responses given in lines 4432-4600 of model_comparison_results.txt

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

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

Task 20: Purpose Inference -> Completion (Python)

Prompt & Responses given in lines 4601-4827 of model_comparison_results.txt

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

Comparison: Select Llama for robust code and detailed comments.

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

Prompt & Responses given in lines 4831-5504 of model_comparison_results.txt

Analysis: GPT-4o and Llama identified and addressed issues in the original script, such as hardcoded paths, lack of error handling, and potential division-by-zero errors. In Zero-Shot, GPT-4o provided a refactored script with robust error handling, command-line argument support via argparse, and documentation. Llama's response also added error handling and a skip_header option, but retained a hardcoded filepath and column index. In Few-Shot, GPT-4o summarized the provided improvements (e.g., type hints, command-line support) and proposed a refactored script with csv.Sniffer for dialect detection, though it was incomplete (cut off mid-code). Llama's few-shot response analyzed the examples, suggesting enhancements like pandas or unit tests, but did not provide a refactored script, missing the core task requirement. Neither model made critical errors, but GPT-4o's incomplete scripts and Llama's lack of a few-shot implementation were notable shortcomings.

Comparison: GPT-4o performed better overall, in zero-shot, for delivering a nearly complete, refactoring.

Task 22: Code Completion and Robustness Enhancement (Python)

Prompt & Responses given in lines 5510-6253 of model_comparison_results.txt

Analysis: Both GPT-4o and Llama successfully completed the clean_line function and enhanced the script, but their approaches and robustness varied. In Zero-Shot, GPT-4o implemented clean_line using str.maketrans to remove punctuation, converted text to lowercase, and trimmed whitespace, maintaining simplicity but lacking error handling or input flexibility. Llama's response was similar. In Few-Shot, GPT-4o provided a detailed summary of the provided examples, highlighting features like UTF-8 encoding and Counter usage, but did not produce a completed script. Llama's few-shot response delivered two versions: a simple one mirroring Example 1 with command-line support and error handling, and a robust one incorporating Example 2's features like encoding fallbacks, Counter for top words, and empty line skipping.

Comparison: Llama performed better overall, particularly in few-shot, for its thorough integration of robust features and complete code delivery.

Summary Comparison

Model Performance:

GPT-4o consistently produced concise, reliable, and readable solutions, often leaning on libraries and idiomatic constructs. Llama 3.1 tended to be more verbose and exploratory, sometimes favoring lower-level implementations. While both models performed well, GPT-4o more frequently delivered production-ready code, especially when paired with well-structured prompts.

Prompting Techniques:

Few-shot prompting improved output quality across both models. It encouraged clearer structure, better error handling, and adherence to the desired output format. Zero-shot prompts were effective for simple tasks but often lacked robustness. Example-driven prompts had the greatest impact in tasks involving formatting, validation, and code generation.

Conclusion:

GPT-4o is better suited for high-quality, efficient code generation with minimal prompting. Llama 3.1 shows promise and excels in providing in-depth explanations, making it useful for learning and exploratory use. Prompting style plays a critical role in guiding both models toward higher accuracy and completeness, particularly in real-world development contexts.