Exploring the Capabilities of Cohere and Gemini for Source Code Summarization Tasks

Sarthak Darji
M.Tech ICT (Software Systems), DA-IICT
Email: 202411030@daiict.ac.in.

Abstract*—Natural language summaries of source code, a process known as source code summarization, significantly contribute to improved code understanding and developer productivity. This research explores the effectiveness of large language models (LLMs), specifically Cohere's `command` and Google's Gemini-Pro, in automating this summarization task. We introduce and assess five different prompting strategies: zero-shot, few-shot, chain-of-thought, critique-driven, and expert-informed prompting. These techniques are applied to Python, JavaScript, and Java code, utilizing standard datasets from the CodeXGLUE benchmark. To evaluate the generated summaries, we employ established metrics such as BLEU, ROUGE-L, METEOR, and BERTScore, which assess both linguistic quality and semantic similarity to reference summaries. Additionally, we perform statistical significance tests to ensure the reliability of observed performance variations across models and prompting approaches. Our results demonstrate that the selection of the prompting method significantly impacts the quality of the generated summaries, with each LLM showing particular advantages depending on the situation. Finally, we provide practical recommendations for prompt design and suggest avenues for future research, including integration with integrated development environments (IDEs), reverse summarization processes, and methods for identifying inaccuracies in generated summaries.

I. INTRODUCTION

One of the most challenging aspects of modern software development is comprehending existing codebases. Developers often dedicate more time to analyzing and interpreting code than to writing new functionalities, particularly when dealing with unfamiliar, legacy, or poorly documented systems. As software applications grow in scale and complexity, the need for intelligent tools that facilitate code understanding becomes increasingly critical.

Source code summarization, the process of generating concise natural language explanations for code elements (such as functions or methods), has emerged as a valuable technique for enhancing code readability and simplifying maintenance [**?**]. These summaries enable developers to quickly grasp the functionality of a code segment without needing to examine its detailed implementation. This capability is especially beneficial in collaborative projects, during the onboarding of new team members, and in the context of code reviews.

Recent advancements in Natural Language Processing (NLP), particularly the development of Large Language Models (LLMs) like GPT, Cohere, and Gemini, have significantly advanced the feasibility of automated code summarization. Trained on vast amounts of both textual and code data, these models can interpret programming constructs and generate relevant descriptions across diverse programming languages.

However, achieving the full potential of LLMs in this area is not trivial. The design and structure of the input provided to the model—commonly known as *prompt engineering*—have a substantial impact on the quality of the generated output. Despite their strong generalization abilities, LLMs are highly sensitive to the way tasks are presented, including the provision of examples, contextual information, and specific instructions.

This research investigates how different prompting methodologies influence the quality of summaries generated by LLMs. We experiment with five distinct strategies—zero-shot, few-shot, chain-of-thought, critique-based, and expert-guided prompting—applied to three prevalent programming languages: Python, JavaScript, and Java. We evaluate performance using established benchmarks from the CodeXGLUE dataset[2] and assess the results using BLEU[3], METEOR[4], ROUGE-L[5], and BERTScore[6] metrics.

Our aim is to gain practical insights into the most effective ways to utilize LLMs for code summarization and to determine which prompting strategies yield the most coherent, informative, and human-like summaries. This study contributes to the advancement of smarter coding tools, automated documentation systems, and next-generation programming assistants.

II. RELATED WORK

The field of code summarization has witnessed significant advancements with the advent of transformer-based architectures tailored for programming languages. Notable models such as **CodeBERT**[7], **CodeT5**[8], and **GraphCodeBERT** have made substantial contributions by adapting pretraining techniques from natural language processing (NLP) to the realm of source code.

CodeBERT is a BERT-based model designed for the joint processing of programming and natural languages, supporting tasks like code retrieval, translation, and summarization. **CodeT5**, built upon the T5 framework, unifies generation and understanding tasks through an encoder-decoder structure, achieving state-of-the-art results on benchmarks related to code documentation. **GraphCodeBERT** enhances semantic understanding by incorporating data flow graphs, enabling it to capture relationships beyond the syntactic structure of code.

The **CodeXGLUE** benchmark suite has played a crucial role in standardizing evaluation within the code intelligence domain. It provides a collection of datasets and tasks—including code summarization—that facilitate consistent comparisons across different models and methodologies.

In contrast to the aforementioned works, which primarily focus on fine-tuning specialized architectures for code-related tasks, our study adopts a distinct approach by leveraging *general-purpose large language models* (LLMs) accessible through APIs. Specifically, we investigate **Cohere's command model** and **Google's Gemini-Pro**, which were not explicitly trained for code summarization.

A central aspect of our investigation is the impact of **prompting strategies**—a factor that has received comparatively less attention in the context of automated code summarization. We evaluate various prompting techniques, including zero-shot, few-shot, chain-of-thought, critique-based, and expert-style prompts, applied across multiple programming languages and model backends. Unlike prior research that primarily focused on model architecture or training data, our contribution highlights how thoughtful prompt design can influence summarization performance even when employing the same underlying model.

To ensure comparability with existing research and facilitate robust evaluation, we utilize datasets and metrics from CodeXGLUE, including BLEU, ROUGE-L, METEOR, and BERTScore. This allows for meaningful comparisons with previous studies while underscoring novel insights gained from prompt-based interaction with general-purpose LLMs.

III. PROBLEM STATEMENT

The central goal is to generate a brief and insightful natural language summary S that precisely describes the fundamental purpose of a given source code segment C. This research investigates the impact of both the selection of large language models (LLMs) and the design of prompting strategies on the quality and effectiveness of the resulting summaries across various programming languages.

IV. METHODOLOGY

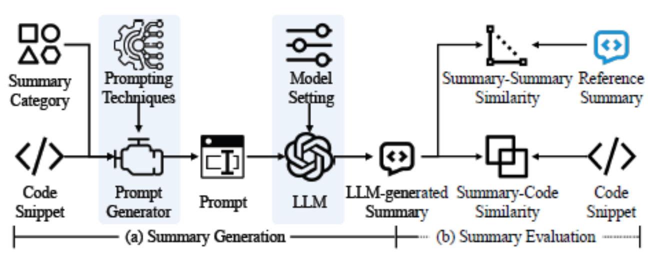

Fig. 1: General workflow of LLM-based code summarization and its effectiveness evaluation

A. Models

Our evaluation focuses on two prominent large language models: Cohere's `command` model and Google's Gemini-Pro. Both models are accessed programmatically via their respective APIs for experimentation.

B. Prompting Techniques

To assess how prompt formulation influences summary quality, we experiment with five distinct prompting strategies. Each strategy aims to guide the LLM in a unique way to enhance contextual understanding and output fluency.

1) Zero-Shot Prompting: The model is asked to summarize the code snippet without being shown any prior examples[9]. This tests the model's intrinsic ability, relying entirely on its pretrained knowledge.

```
Summarize the following function in
one or two sentences:
{code}
```

2) Few-Shot Prompting: Few-shot prompting is also known as in-context learning[10]. Here, before presenting the main code to be summarized, a few sample code-summary pairs are included. This allows the model to learn from examples and mimic the expected output style.

```
Example 1:
def add(a, b): return a + b
Summary: Adds two numbers.

Now summarize:
{code}
```

3) Chain-of-Thought Prompting: Here, the model is guided to break down its reasoning in steps—such as analyzing inputs, understanding function behavior, and then composing a summary. This method is useful for improving the interpretation of more complex logic.

```
Think step by step:
1. Understand what the function
does.
2. Identify inputs and outputs.
3. Summarize the functionality.

Function:
{code}
```

4) Critique Prompting: This approach encourages the model to first generate a summary, then critically review its own output for clarity and completeness, making revisions if needed.

```
Summarize the function below.
Then critique your summary for
completeness and correctness. If
necessary, refine it.

Function:
{code}
```

5) Expert Prompting: The model is instructed to act as a senior software engineer and provide a simplified explanation, intended for junior developers. This strategy prioritizes precision and readability.

```
You are a senior software engineer.
Explain the following function to a
junior developer:
{code}
```

Each of these strategies is applied to all LLMs and programming languages under study, enabling a comparative evaluation of summarization performance across scenarios.

C. Datasets

We employ datasets from the CodeXGLUE benchmark to conduct experiments in Python, JavaScript, and Java. These datasets consist of pairs of code fragments and their respective natural language descriptions (e.g., docstrings), serving as ground truth references.

Dataset Overview:

- **Python:** 251,820 examples featuring utility functions, algorithm implementations, and class methods from open-source repositories.
- **JavaScript:** 58,025 functions, primarily focusing on UI utilities, DOM interaction, and asynchronous logic.
- **Java:** 164,923 samples including constructors, method definitions, and object-oriented patterns.

These datasets are used in our fifth experimental phase, where we assess the performance of different prompting strategies across models and languages.

Data Preprocessing:

- **Length Filtering:** Code-summary pairs were removed if summaries had fewer than three tokens or if the function body had fewer than five lines or more than fifty lines. This helped ensure that examples were neither trivial nor overly complex.
- **Duplicate Removal:** Redundant functions and near-identical variants were filtered out using hash-based fingerprinting techniques to preserve dataset diversity.
- **Formatting Normalization:** Uniform formatting was applied to whitespace, indentation, and docstrings to maintain prompt consistency and reduce parsing variability.
- **Token Limit Control:** Code samples that exceeded reasonable token limits—due to embedded data or excessive literals—were discarded to avoid truncation issues when interfacing with LLM APIs.
- **Stratified Sampling:** For evaluation purposes, we randomly selected 5 code snippets per language, ensuring balanced representation across all five prompting strategies and both models.

While CodeXGLUE provides a robust foundation for benchmarking, its samples are mostly concise and self-contained. Therefore, the findings in this work may not fully generalize to large-scale, multi-file enterprise applications, where broader contextual understanding is required.

V. EXPERIMENTS AND EVALUATION

A. Evaluation Metrics

To gauge the effectiveness of the generated code summaries, we employed four well-established evaluation metrics: BLEU, METEOR, ROUGE-L, and BERTScore. These metrics collectively offer insights into both the lexical and semantic quality of the generated outputs.

BLEU (Bilingual Evaluation Understudy) assesses the degree to which n-gram sequences in the generated summary correspond to those in the reference summary. Initially developed for machine translation, it is commonly used for summarization tasks. BLEU emphasizes precise word matches but might not always recognize semantically similar paraphrases.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) enhances BLEU by considering stemming, synonyms, and word order. It establishes more adaptable alignments between generated and reference texts, allowing for variations in phrasing while still quantifying accuracy.

ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation – Longest Common Subsequence) measures the longest shared sequence of words between two texts, highlighting the extent to which the reference content is retained. This metric focuses on recall and is useful for assessing the fluency and completeness of the generated summaries.

BERTScore compares semantic similarity by utilizing embeddings from pre-trained BERT models. Unlike metrics based on token overlap, it measures the alignment of meaning between the generated and reference texts, even with differences in wording. This makes it particularly suitable for evaluating outputs from large language models, which frequently produce paraphrased yet accurate responses.

By integrating these varied metrics, we ensure a comprehensive evaluation that captures both surface form and underlying meaning. All scores were calculated using standard toolkits and averaged across multiple code samples for each combination of model, language, and prompting strategy.

VI. RESULTS AND DISCUSSION

A. Model Performance

Table I showcases the comparative results of the Cohere and Gemini models evaluated across various prompting strategies and programming languages. Overall, Gemini demonstrated superior performance, particularly in metrics like BLEU and BERTScore. These outcomes suggest that Gemini was more effective at producing summaries with accurate n-gram overlap and stronger semantic alignment with reference summaries. For instance, on Python code, Gemini achieved its best BLEU score (0.08008) and highest BERTScore (0.66984) using the few-shot prompting method. This trend continued across JavaScript and Java, where Gemini consistently surpassed Cohere in semantic evaluation, especially under zero-shot and few-shot scenarios.

Prompt design had a clear influence on performance. Few-shot prompts consistently led to the most accurate summaries across all tested languages and models, indicating that including example pairs helps guide the models to produce better outputs. Prompts designed in expert-mode—offering detailed guidance similar to expert annotations—also yielded competitive results, often just behind few-shot performance.

Conversely, zero-shot prompting yielded the weakest results across most evaluation metrics. This reinforces the idea that LLMs benefit significantly from contextual cues or in-context examples. Prompting strategies such as chain-of-thought and critique prompting achieved moderate scores, implying that while reasoning steps or structured feedback help improve output quality, they may not be as effective as directly demonstrating target behavior through examples.

Among the three programming languages, Python consistently returned the highest metric scores. This could be attributed to Python's cleaner syntax and the common presence of structured documentation. JavaScript summaries, particularly from Cohere, showed low BLEU scores, likely due to the language's dynamic nature and asynchronous patterns. Java results were somewhat mixed but improved notably when processed with Gemini, particularly in metrics assessing semantic quality like BERTScore.

Overall, the analysis highlights the importance of both model selection and thoughtful prompt engineering in enhancing summarization outcomes. The findings also stress that strategic prompting is essential to fully harness the capabilities of LLMs for code-related tasks.

B. Limitations

While this research provides insightful findings on the performance of LLMs in code summarization, several limitations might influence the generalizability of these results:

- **Accuracy Issues in Summaries:** Occasionally, particularly with zero-shot or expert-style prompts, the models produced summaries that inaccurately described the code's functionality. This issue of hallucination was more prevalent with abstract or inadequately documented functions.
- **Dataset Representativeness:** The datasets employed—including CodeXGLUE and CodeSearchNet—primarily consist of short, isolated functions. This might not fully reflect real-world software ecosystems, which often involve extensive, interconnected modules.
- **Resource Constraints:** This study was limited by available hardware and API access. Consequently, we restricted the scope to three programming languages and four LLMs, each evaluated with five prompting techniques. Larger models such as GPT-4 and Gemini 1.5 Pro, or open-source options like StarCoder, were not examined due to these limitations.
- **Prompt Sensitivity:** Certain LLMs showed considerable fluctuation in output quality with minor alterations to prompt wording or structure. This sensitivity introduces variability and underscores the difficulty of ensuring consistent results across different executions.
- **Limited Evaluation Scope:** Our assessment relied on four metrics—BLEU, ROUGE-L, METEOR, and BERTScore—which, while informative, do not fully encompass the functional correctness or practical value of the summaries. Human evaluations or task-based assessments (e.g., how summaries assist in debugging) could offer a more comprehensive evaluation.

C. Model Performance Comparison: Cohere

To gain deeper insight into how Cohere performs across different programming languages and prompting methods, we present a visual analysis through a bar chart. This chart illustrates how the model scored on four key evaluation metrics—BLEU, METEOR, ROUGE-L, and BERTScore—across three programming languages (Python, JavaScript, and Java) and five distinct prompting styles: Zero-shot, Few-shot, Chain-of-thought, Critique, and Expert.

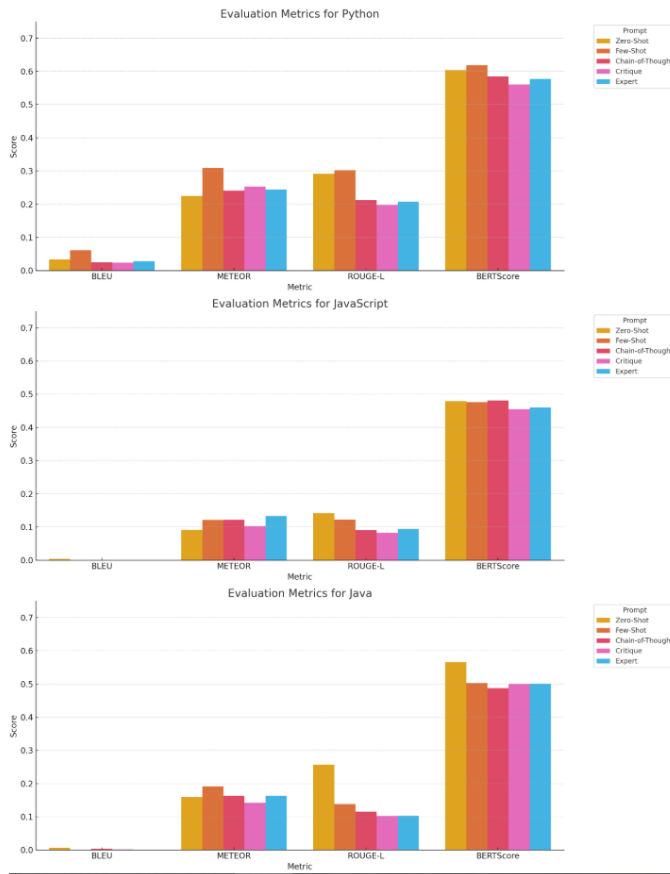

Fig. 1: Performance of Cohere LLM using various prompting strategies across Python, JavaScript, and Java using BLEU, METEOR, ROUGE-L, and BERTScore.

The visualization indicates that the Few-shot and Expert prompting methods tend to produce stronger performance, particularly in Python-related tasks. These strategies often resulted in more accurate and contextually relevant summaries. In contrast, Java and JavaScript showed generally lower scores, though Expert prompting still helped improve semantic alignment, as reflected in elevated BERTScore values.

D. Model Performance Comparison: Gemini

To evaluate Gemini's performance across different languages and prompting methods, we present a bar chart showcasing the evaluation metrics—BLEU, METEOR, ROUGE-L, and BERTScore—across Python, JavaScript, and Java. The chart illustrates the results for five distinct prompting

TABLE I: Evaluation scores for different LLM models and prompting techniques across programming languages using text and semantic similarity metrics

Sr No.	LLM Model	Prompting Technique	Python				JavaScript				Java			
			Text Similarity			Semantic Similarity	Text Similarity			Semantic Similarity	Text Similarity			Semantic Similarity
			BLEU	METEOR	ROUGE-L	BERTScore	BLEU	METEOR	ROUGE-L	BERTScore	BLEU	METEOR	ROUGE-L	BERTScore
1	Cohere	Zero-Shot	0.03236	0.25776	0.3128	0.60324	0	0.08502	0.12186	0.42916	0	0.17964	0.20172	0.55048
		Few-Shot	0.04222	0.31954	0.27306	0.56674	0	0.12754	0.10324	0.44608	0	0.18714	0.12962	0.4999
		Chain-of-Thought	0.03252	0.2486	0.23488	0.58476	0	0.12532	0.0981	0.47526	0.0044	0.15268	0.1224	0.4301
		Critique	0.03056	0.26784	0.22194	0.55578	0	0.10092	0.09452	0.47544	0	0.14062	0.12172	0.51728
		Expert	0.0403	0.2616	0.25468	0.56874	0	0.14426	0.09968	0.4466	0	0.15354	0.10298	0.47714
2	Gemini	Zero-Shot	0.03428	0.19126	0.26934	0.6041	0.008	0.09656	0.16156	0.52874	0.01314	0.13838	0.31186	0.58028
		Few-Shot	0.08008	0.29794	0.33116	0.66984	0	0.11442	0.14134	0.50674	0	0.19624	0.14636	0.50596
		Chain-of-Thought	0.01564	0.23324	0.18874	0.58426	0.00166	0.11772	0.08286	0.48702	0.00268	0.17304	0.10814	0.54388
		Critique	0.01592	0.23902	0.17224	0.56468	0	0.1054	0.0712	0.43436	0.0028	0.1424	0.0828	0.48292
		Expert	0.01544	0.22646	0.16008	0.5856	0	0.12166	0.08796	0.47356	0	0.17208	0.10228	0.52444

strategies: Zero-shot, Few-shot, Chain-of-thought, Critique, and Expert.

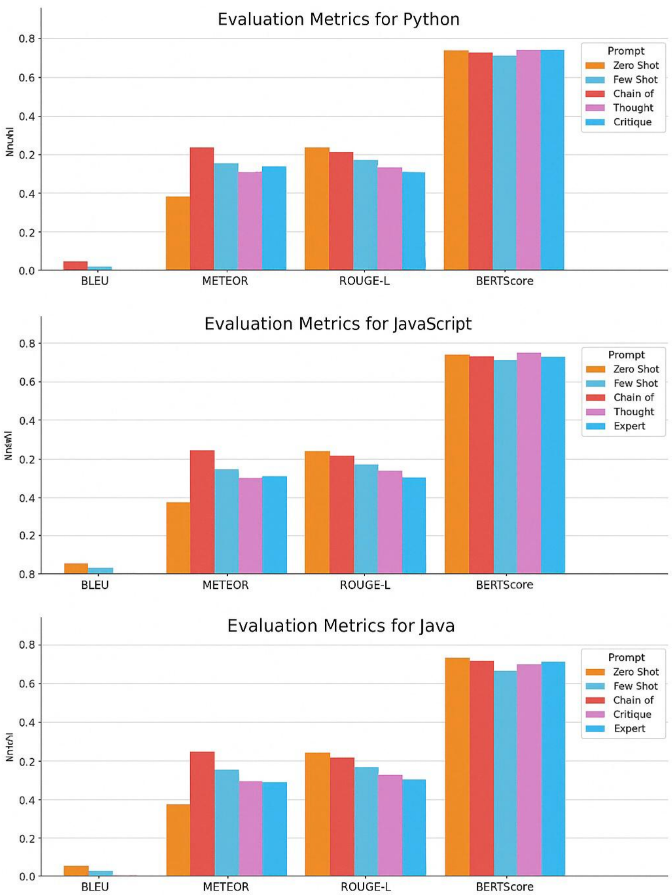

Fig. 2: Performance of Gemini LLM using various prompting strategies across Python, JavaScript, and Java using BLEU, METEOR, ROUGE-L, and BERTScore.

The Few-shot prompting approach yielded the strongest performance across all metrics, particularly for Python. Although the results for Java and JavaScript were comparatively lower, Few-shot prompting still provided superior semantic alignment, reflected in higher BERTScores. Zero-shot prompting performed poorly across all languages, while Chain-of-thought and Critique methods showed moderate results, particularly for Java and JavaScript, where BLEU and METEOR scores remained low. The Expert approach, while similar to Few-shot in some areas, did not show significant improvement in certain cases, especially for Java.

VII. CONCLUSION

In summary, the design of effective prompts is a crucial factor in enhancing the quality of code summarization. Our findings underscore the significance of carefully selecting prompting strategies to produce high-quality summaries.

VIII. LIMITATIONS AND FUTURE DIRECTIONS

- **Automated Code Comment Generation Tool:** *Explanation:* While this study primarily investigates general-purpose code summarization across different programming languages using prevalent LLMs, consistent and clear code comments are equally important in practical software development. Our aim is to develop a tool that automatically generates editable comments above functions or methods as developers write code. These comments will be designed to be easily understandable, making them particularly beneficial for novice developers.
- **Reverse Code Summarization (Natural Language to Code):** *Explanation:* Although our research focuses on summarizing code into natural language (code $\rightarrow$ text), the reverse process, generating code from natural language descriptions (text $\rightarrow$ code), is also crucial. This approach is vital for:
  - Enabling code generation driven by natural language.
  - Improving intelligent coding assistants (e.g., GitHub Copilot, Amazon CodeWhisperer).

  Future Goal: Explore the capabilities of models like Cohere and Gemini when tasked with converting natural language summaries into accurate and complete code segments.
- **Framework for Evaluating Factual Accuracy and Hallucination:** *Explanation:* We intend to develop a metric or evaluation framework that assesses the factual correctness of code summaries generated by LLMs, with a specific focus on identifying instances of hallucinated or incorrect information. *Why it's Important:* Current metrics such as BLEU and BERTScore measure fluency but do not evaluate the factual accuracy of the summaries. A framework designed to assess factuality can significantly improve the reliability of LLM-generated code summaries.

REFERENCES

1) S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, “The effect of modularization and comments on program comprehension,” in Proceedings of the 5th International Conference on Software Engineering. San Diego, California, USA: IEEE Computer Society, March 9-12 1981, pp. 215–223.
2) S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” in Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, virtual, December 2021, pp. 1–14.
3) K. Papineni, S. Roukos, T. Ward, and W. Zhu, “BLEU: A method for automatic evaluation of machine translation,” in Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia, PA, USA: ACL, July 6-12 2002, pp. 311–318.
4) S. Banerjee and A. Lavie, “METEOR: an automatic metric for MT evaluation with improved correlation with human judgments,” in Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization. Ann Arbor, Michigan, USA: Association for Computational Linguistics, June 29 2005, pp. 65–72.
5) C.-Y. Lin, “ROUGE: A package for automatic evaluation of summaries,” in Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics – workshop on Text Summarization Branches Out. Barcelona, Spain: Association for Computational Linguistics, July 21-26 2004, pp. 74–81.
6) S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Semantic similarity metrics for evaluating source code summarization,” in Proceedings of the 30th International Conference on Program Comprehension. Virtual Event: ACM, May 16-17 2022, pp. 36–47.
7) Feng, Z., et al. ”CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” EMNLP 2020.
8) Wang, Y., et al. ”CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.” EMNLP 2021.
9) W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang, H. Qian, Y. Liu, and Z. Chen, “Automatic code summarization via chatgpt: How far are we?” CoRR, vol. abs/2305.12865, pp. 1–13, 2023.
10) M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, “Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning,” in Proceedings of the 46th International Conference on Software Engineering. Lisbon, Portugal: ACM, April 14-20 2024, pp. 39:1–39:13.

APPENDIX

A. Sample Input and Summaries

Below is an example of a **Python** function used in our evaluation, followed by the reference summary and outputs from each prompting strategy using the **Gemini** model.

Input Code Snippet:

Listing 1: Example Function Used for Summarization

```
def postappend(self):
    if not self.doc and self.parent.doc:
        self.setdocument(self.parent.doc)

    if self.doc and self.doc.deepvalidation:
        self.deepvalidation()
```

Reference Summary:

This method will be called after an element is added to another and does some checks. It can do extra checks and if necessary raise exceptions to prevent addition. By default makes sure the right document is associated.This method is mostly for internal use.

LLM-Generated Summaries (Gemini):

1) Zero-Shot:

This function ensures document association and performs optional deep validation after an element is appended.

2) Few-Shot:

This method is called after an element is added to another. It checks and associates the element with the correct document, and performs deep validation if necessary. It’s primarily for internal use and can raise exceptions to prevent addition if checks fail.

3) Chain-of-Thought:

Okay, let’s break down the ‘postappend‘ function step-by-step.

1. Understanding the Function’s Purpose

The docstring states that this function is called *after* an element has been added as a child to another element (its parent). It primarily serves as a check and potential validation step triggered by the act of appending. The core functions it performs are:

- Document Association: Ensures that if the newly added element (‘self‘) doesn’t yet have an associated document (‘self.doc‘) but its parent element does (‘self.parent.doc‘), then the newly added element and its children are associated with the parent’s document. - Deep Validation: If the element *does* have an associated document and that document has ”deep validation” enabled, then a deeper validation process (‘self.deepvalidation()‘) is triggered.

Essentially, it handles document propagation and potentially triggers a validation process upon the child’s addition. It exists mostly for internal use.

2. Inputs and Outputs

- Inputs: - ‘self‘: Represents the element that was *just* added as a child to another element. This element has attributes ‘doc‘ (the associated document) and ‘parent‘ (the element it was added to). - Implicit input: ‘self.doc.deepvalidation‘ (boolean value indicating if deep validation should be performed for the associated document)
- Outputs: - No explicit return value (implicitly returns ‘None‘). - Side Effects: - May set the ‘doc‘ attribute of ‘self‘ and its descendants if they don’t have one and the parent does. - May trigger ‘self.deepvalidation()‘ if conditions are met. - The ‘deepvalidation‘ method itself could potentially raise exceptions if validation fails.
3. Summary
The ‘postappend‘ function is a method called after an element is appended as a child to another element. It checks if the newly added element needs to be associated with the same document as its parent. If so, it propagates the document association down the child’s lineage. Furthermore, if the associated document has deep validation enabled, this function triggers a deep validation process for the newly added element. This is mostly an internal method meant for validation and document consistency.

4) Critique:

Okay, here’s the summary:
Summary:
The ‘postappend‘ method is called after an element is added as a child to another element within a document structure. Its primary purpose is to perform post-addition checks and actions. By default, it ensures that the newly appended element and its children are correctly associated with the parent element’s document. If deep validation is enabled on the document, it triggers a deep validation check on the element. It’s intended primarily for internal use and can be overridden to implement custom validation and association logic. Furthermore it can raise exceptions to prevent addition.
Critique and Refinement:
My initial summary captures the main points, but it could be slightly improved for clarity and completeness. Here’s a breakdown of what’s good and what’s missing:
- Good: - Identifies the triggering event (element addition). - Highlights the document association aspect. - Mentions deep validation. - Indicates its internal usage.
- Missing/Could be Improved: - Emphasis on potential for custom validation: While I say it can be overridden, I could be more explicit about the fact that this method *allows* the developer to implement their own checks/validation logic to control how an element gets added. - Error Handling (Raising Exceptions): The original docstring explicitly mentions the potential for raising exceptions to *prevent* addition. This is a crucial aspect of the method’s functionality and should be included.
Revised Summary:
The ‘postappend‘ method is an internal function called after an element is added as a child to another element within a document structure. Its default behavior is to ensure that the newly appended element and its children are correctly associated with the parent element’s document. If deep validation is enabled on the document, it triggers a deep validation check on the element. Importantly, this method provides a hook for implementing custom validation and checks. Developers can override this method to enforce specific constraints on element addition and can raise exceptions to prevent the element from being added if the validation fails.

5) Expert:

Okay, here’s the breakdown of the ‘postappend‘ function for a junior developer:
”The ‘postappend‘ function is called *after* an element is added as a child to another element. It’s mainly for internal use.
Think of it like a final sanity check. Its primary jobs are:
1. **Document Association:** If the newly added element doesn’t have a document associated with it yet, but its parent *does*, it automatically sets the element’s document to be the same as the parent’s. This ensures that all elements in the tree belong to the correct document. The ‘setdocument‘ method recursively sets the document for this element and all its children.
2. **Deep Validation:** If the document associated with the element has ”deep validation” enabled, it calls the ‘deepvalidation‘ method. This performs a thorough check of the element’s validity within the context of the document.
Essentially, ‘postappend‘ makes sure the element is properly integrated into the document structure and valid before moving on.”

Below is an example of a **Javascript** function used in our evaluation, followed by the reference summary and outputs from each prompting strategy using the **Cohere** model.

Input Code Snippet:

Listing 2: Example Function Used for Summarization

```
function inRange(value, min, max) {
  const int = parseInt(value, 10);

  return (
    `${int}` === `${value.replace(/^0/, '')}` &&
```

```
        int >= min &&
        int <= max
    );
}
```

Reference Summary:

Determine if value is within a numeric range@param String—Number value - value to check@param Number min - start of range (inclusive)@param Number max - end of range (inclusive)@returns Boolean whether or not value is within range

LLM-Generated Summaries (Cohere):

1) **Zero-Shot:**

The inRange function checks if a given value is an integer within a specified range, ensuring that leading zeros are ignored.

2) **Few-Shot:**

The function ‘inRange‘ takes three parameters: ‘value‘, ‘min‘, and ‘max‘. It first converts the ‘value‘ to an integer by removing any leading zeros and parsing it as a decimal number. The function then checks if the resulting integer representation of the ‘value‘ is within the range specified by ‘min‘ and ‘max‘. If the integer value is within the range, the function returns ‘true‘; otherwise, it returns ‘false‘. In summary, this function determines whether a given value, after being converted to an integer, falls within a specified numeric range.

3) **Chain-of-Thought:**

Here is a step-by-step explanation of the function:
1. Understanding the Function: The function is named ”inRange,” and it appears to be a utility function that checks if a given ”value” is within a specified ”min” and ”max” range.
2. Identifying Inputs and Outputs: - Inputs: - ”value”: This is the input value that the function will check if it falls within the specified range. It is expected to be a string representation of a number. - ”min”: The minimum value of the range. - ”max”: The maximum value of the range.
- Output: - The function returns a boolean value (”true” or ”false”) indicating whether the ”value” is within the specified range, including whether it is a valid number.
3. Summarization: The ”inRange” function takes a ”value,” ”min,” and ”max” as inputs. It first converts the ”value” to an integer using ”parseInt” with a radix of 10. Then, it performs the following checks:
- It ensures that the converted integer value is the same as the original value without any leading zeros.
- It checks if the integer value is greater than or equal to the ”min” value. - It checks if the integer value is less than or equal to the ”max” value.
If all these conditions are met, the function returns ”true,” indicating that the ”value” is within the specified range and is a valid number. Otherwise, it returns ”false.”
In summary, the ”inRange” function validates whether a given ”value” is a number within a specified range, and it returns a boolean indicating the result of this validation.

4) **Critique:**

The function ‘inRange‘ takes three arguments: ‘value‘, ‘min‘, and ‘max‘. It first converts the ‘value‘ to an integer using ‘parseInt‘ with a radix of 10. It then checks if the integer representation of the ‘value‘ is the same as the ‘value‘ with any leading zeros removed. This ensures that the ‘value‘ is a valid integer without any extra leading zeros. Finally, it checks if the integer is within the range specified by ‘min‘ and ‘max‘. If all these conditions are met, the function returns ‘true‘, indicating that the ‘value‘ is within the specified range. Otherwise, it returns ‘false‘.
Critique: The summary provides a good overview of the function’s purpose and the checks it performs. However, it could be improved by mentioning the data types of the arguments and the return value of the function. Here’s a refined version:
The function ‘inRange‘ takes three arguments: ‘value‘ (a string or number), ‘min‘ (a number), and ‘max‘ (a number). It first converts the ‘value‘ to an integer using ‘parseInt‘ with a radix of 10. It then checks if the integer representation of the ‘value‘ matches the ‘value‘ with any leading zeros removed, ensuring it is a valid integer. Finally, it verifies if the integer falls within the range defined by ‘min‘ and ‘max‘. The function returns ‘true‘ if all conditions are satisfied, indicating that the ‘value‘ is within the specified range, and ‘false‘ otherwise. The return type of the function is a boolean.

5) **Expert:**

Here’s a concise summary of the ‘inRange‘ function for a junior developer:
The ‘inRange‘ function checks if a given ‘value‘ is within a specified range defined by ‘min‘ and ‘max‘. It first converts the ‘value‘ to an integer using ‘parseInt‘, removing any leading zeros. Then, it verifies if the resulting integer representation is equal to the original value without leading zeros, and if it falls within the specified range. If both conditions are met, the function returns ‘true‘, indicating that the value is within the valid range. Otherwise, it returns ‘false‘.
The function effectively ensures that the value is a valid integer representation and that it lies between the minimum and maximum values provided.

Below is an example of a **Java** function used in our evaluation, followed by the reference summary and outputs from each prompting strategy using the **Gemini** model.

Input Code Snippet:

Listing 3: Example Function Used for Summarization

```
public <T> T getNodeMetaData(Object key) {
    if (metaDataMap == null) {
        return (T) null;
    }
    return (T) metaDataMap.get(key);
}
```

Reference Summary:

Gets the node meta data.@param key - the meta data key @return the node meta data value for this key

LLM-Generated Summaries (Gemini):

1) Zero-Shot:

Retrieves metadata associated with the given key, returning null if no metadata exists or the map is null.

2) Few-Shot:

This function retrieves metadata associated with a given key from an internal map. If the map is empty or the key is not found, it returns null. The returned value is cast to the specified type ‘T‘.

3) Chain-of-Thought:

Okay, let’s break down this Java function step-by-step:
1. Understanding what the function does:
The function ‘getNodeMetaData‘ retrieves metadata associated with a given key from a map. It appears that the metadata is stored in a map called ‘metaDataMap‘. The function is generic, meaning it can return metadata of any type ‘T‘. It handles the case where the map is empty or doesn’t exist by returning ‘null‘.
2. Identifying the inputs and outputs:
- Input: - ‘key‘ (Object): The key used to retrieve the metadata from the map. The type is ‘Object‘, indicating that keys of any type can be used.
- Output: - ‘T‘ (Generic Type): The metadata associated with the provided key. The type of the metadata is determined by the generic type ‘T‘. - ‘null‘ (T): If ‘metaDataMap‘ is ‘null‘ or there’s no value associated with the key, the function returns ‘null‘. This ‘null‘ is cast to type ‘T‘.
3. Summary:
The ‘getNodeMetaData‘ function retrieves metadata from a map (‘metaDataMap‘) using a given key. If the map is null or the key doesn’t exist, it returns null. The function is generic, allowing it to return metadata of any type specified by the caller. In essence, it’s a type-safe wrapper around a map lookup for metadata.

4) Critique:

Okay, here’s the summary:
Summary:
The ‘getNodeMetaData‘ function retrieves metadata associated with a given key from an internal metadata map. If the metadata map is null, it returns null. Otherwise, it returns the value associated with the key, cast to the generic type ‘T‘.
Self-Critique and Refinement:
The initial summary is functional but could be improved. Here’s what’s missing and how to refine it:
* **Missing Context:** The initial summary doesn’t convey *why* this function might be useful or *what kind* of metadata we’re dealing with. It’s a low-level technical description. We should hint at the function’s purpose within a larger context (though without knowing the complete surrounding code, this will be somewhat speculative). * **Null Handling:** While the summary mentions null handling, it could be more explicit.
Here’s a refined summary:
Refined Summary:
The ‘getNodeMetaData‘ function retrieves metadata associated with a specific key for a node (likely within a data structure or system). It checks if an internal metadata map exists. If the map is null (implying no metadata has been stored), the function returns null. Otherwise, it retrieves the value corresponding to the provided key from the map and casts it to the expected type ‘T‘. This allows accessing node-specific properties or configuration settings dynamically. If no metadata is available, this function will return ‘null‘.
Explanation of Changes:
* **”for a node (likely within a data structure or system)”**: This adds context. The name ”getNodeMetaData” strongly suggests it’s part of a system dealing with nodes. * **”implying no metadata has been stored”**: Clarifies the implication of the null map, giving more context. * **”This allows accessing node-specific properties or configuration settings dynamically.”**: Hints at a potential use case of the function. * **”If no metadata is available, this function will return ‘null‘.”**: Provides a more straightforward statement.
The refined summary is better because it provides more context and hints at the purpose of the function within a larger system. While still limited by the lack of surrounding code, it’s a more informative and useful description.

5) Expert:

Okay, here’s the summary:
”This Java function ‘getNodeMetaData‘ retrieves metadata associated with a given ‘key‘. It checks if the ‘metaDataMap‘ (presumably a ‘Map‘ holding

the metadata) is initialized. If it's 'null' (meaning it hasn't been created or initialized), it returns 'null'. Otherwise, it retrieves the value associated with the provided 'key' from the 'metaDataMap' and returns it, casting it to the type 'T'. Effectively, it's a getter method for metadata stored in a map, handling the case where the map might not exist."

This samples illustrates how different prompting techniques influence the granularity, clarity, and correctness of the generated summaries.