CODE SUMMARIZATION AND EVALUATION USING DEEPSEEK AND LLAMA

Tilak Brahmbhatt
M.Tech ICT (Software Systems), DAU
Email: 202411016@daiict.ac.in

***Abstract*—Code summarization, which involves generating natural language descriptions of code, is vital for improving code comprehension and boosting developer productivity. This paper explores the effectiveness of large language models (LLMs), specifically DeepSeek-Coder R1.5B and LLaMA 3.2B models, for automating code summarization tasks. We introduce and assess five distinct prompting strategies: zero-shot, few-shot, chain-of-thought, critique-driven, and expert-guided prompts. These strategies are tested on code written in Python, JavaScript, and Java, utilizing datasets from the CodeXGLUE benchmark. Model outputs are evaluated using standard metrics like BLEU, ROUGE-L, METEOR, and BERTScore to measure linguistic quality and semantic coherence with reference summaries. In addition, statistical tests are performed to verify the significance of performance differences between models and methods. Our results demonstrate that the choice of prompting technique significantly affects summary quality, with each LLM showing unique strengths in different scenarios. We conclude by providing practical recommendations for prompt design and suggest potential areas for future work, including domain-specific code summarization, reverse summarization (summary-to-code), and methods for detecting hallucinations in generated summaries.**

***Index Terms*—Code Summarization, Prompt Engineering, Large Language Models, Evaluation Metrics**

I. INTRODUCTION

Understanding existing code remains one of the most challenging tasks in contemporary software development. Developers often dedicate more time to reading and interpreting code than to writing new functionality—especially when dealing with legacy systems, unfamiliar modules, or poorly documented projects. As software systems grow in complexity and scale, the need for intelligent solutions that aid in code understanding becomes increasingly critical.

Code summarization, the process of generating concise natural language descriptions for code components like functions or methods—has emerged as a practical approach to improve code clarity and ease software maintenance [1]. These summaries offer developers a quick overview of code behavior, eliminating the need to analyze every line in detail. Such tools prove valuable in team collaborations, onboarding scenarios, and during peer reviews.

With the advancement of Natural Language Processing (NLP), particularly through the emergence of Large Language Models (LLMs) such as GPT, DeepSeek, LLaMA, Cohere, and Gemini, the feasibility of automating code summarization has significantly improved. These models, trained on vast datasets comprising both programming and natural languages, are capable of parsing code semantics and generating meaningful summaries across multiple languages.

However, leveraging the full potential of LLMs in this domain is a non-trivial task. The structure and formulation of the input—referred to as prompt engineering—play a crucial role in shaping the quality of the generated output. Despite their broad generalization capabilities, LLMs are particularly sensitive to the way instructions, examples, and context are presented.

This work investigates how various prompting techniques influence the summary generation performance of LLMs. We experiment with five distinct strategies—zero-shot, few-shot, chain-of-thought, critique-driven, and expert-informed prompting—applied to source code written in Python, JavaScript, and Java. To evaluate the outcomes, we use well-established benchmarks from the CodeXGLUE dataset [2] and compute performance using BLEU [3], METEOR [4], ROUGE-L [5], and BERTScore [6]. These metrics allow us to assess both surface-level similarity and deeper semantic alignment between model-generated summaries and human-written references.

The goal of this study is to derive actionable insights into prompt design and to determine which strategies yield the most accurate, readable, and human-aligned code summaries. Our findings aim to support the development of advanced tooling for intelligent code assistance, documentation automation, and AI-powered programming environments.

II. RELATED WORK

Recent advancements in code summarization have been largely driven by transformer-based architectures tailored for programming tasks. Models like CodeBERT[7], CodeT5[8], and GraphCodeBERT have significantly advanced this domain by adapting NLP-centric pretraining techniques to source code understanding.

CodeBERT modifies the original BERT model to jointly encode both code and natural language, enabling it to perform tasks such as code search, translation, and summarization. **CodeT5**, an extension of the T5 architecture, unifies generation and comprehension tasks within an encoder-decoder structure, achieving top-tier results on code documentation benchmarks. In contrast, **GraphCodeBERT** enhances its learning by integrating data flow graphs, which helps capture deeper semantic relationships beyond syntactic analysis.

The **CodeXGLUE benchmark** suite has been instrumental in unifying evaluation methodologies within the field of code intelligence. It offers a diverse set of datasets and tasks that span multiple programming languages, facilitating objective and repeatable model assessments. Within this suite, the code summarization dataset is a commonly used standard for evaluating the capability of models to convert code into natural language explanations. It includes annotated pairs of code snippets and their human-written summaries, making it ideal for supervised training and benchmarking.

Where most previous studies have focused on fine-tuned models developed specifically for programming-related tasks, our research follows a distinct direction by using general-purpose large language models (LLMs) running locally. In particular, we examine the performance of **DeepSeek-Coder R1.5B** and **LLaMA 3.2B**, both executed through the **Ollama framework**. These models are not fine-tuned for code summarization, which allows us to evaluate their out-of-the-box performance in a cost-effective and customizable environment, without relying on commercial APIs.

The central theme of our investigation is prompt design—an area that has received comparatively less attention in automated code summarization literature. We evaluate five different **prompting strategies**: zero-shot, few-shot, chain-of-thought, critique-driven, and expert-informed prompts, across multiple programming languages and LLM backends. Unlike prior work that mainly focuses on model architecture and training data, our study emphasizes the influence of prompt formulation on the summarization output, even with the same base model.

To ensure compatibility with earlier research and to maintain evaluation consistency, we adopt the datasets and scoring metrics provided by CodeXGLUE. These include BLEU, ROUGE-L, METEOR, and BERTScore, which collectively allow us to evaluate both textual overlap and semantic similarity between generated and reference summaries. This alignment supports meaningful comparisons with existing models and also brings forth new observations stemming from prompt-based interactions with general-purpose LLMs.

III. PROBLEM STATEMENT

Given a source code snippet C, the goal is to produce a brief, informative natural language summary S that faithfully captures the primary functionality and purpose of C. This study explores how two central variables influence the quality of such summaries: **(1)** the type of large language model (LLM) used, and **(2)** the design and structure of the prompts given to the model. To examine the adaptability of this approach, we test across three programming languages: Python, JavaScript, and Java.

To determine the effectiveness and accuracy of the generated summaries, each output is compared against a corresponding reference summary written by a human expert. The comparison is conducted using well-established metrics such as BLEU, ROUGE-L, METEOR, and BERTScore, which collectively assess linguistic quality and semantic similarity. This evaluation framework ensures that the generated summaries are not only grammatically sound but also faithful to the functional meaning of the source code.

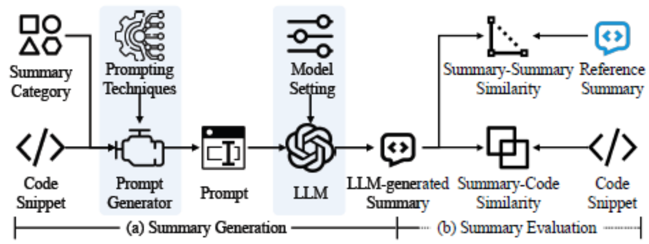

Fig. 1: General workflow of LLM-based code summarization and its effectiveness evaluation

IV. METHODOLOGY

A. Models

Our evaluation focuses on two prominent large language models: DeepSeek-Coder R1.5B and LLaMA 3.2B. As both models are deployed on a local machine using the Ollama framework, they can be accessed programmatically for experimentation.

B. Prompting Techniques

To evaluate how the design of prompts affects summary generation, we implement five distinct prompting techniques. Each technique guides the LLM differently, aiming to enhance the model's comprehension of the code context and improve the fluency and accuracy of the generated summaries.

1) Zero-Shot Prompting: In this method, the model is prompted to summarize the code fragment without seeing any examples beforehand [9]. This setup relies entirely on the model's pretraining to understand and explain the code.

```
Given the following function,
summarize it in one or two
sentences:
{code}
```

2) Few-Shot Prompting: Also known as in-context learning [10], this approach provides the model with several code-summary examples before presenting the actual code to be summarized. These examples serve as references for the desired output style and structure.

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

Now summarize:
{code}
```

3) Chain-of-Thought Prompting: This strategy instructs the model to reason through the function in a structured sequence—identifying inputs, outputs, and operations—before generating the final summary. It is particularly useful for summarizing code with intricate logic.

```
Try thinking step by step:
1. Have understanding of the
function.
2. Based on that, recognize inputs
```

```
and outputs.
3. Then, summarize the
functionality.

Function:
{code}
```

4) Critique Prompting: In this method, the model is asked to first generate a summary, then evaluate its own output for accuracy and completeness. If any gaps or ambiguities are found, it is instructed to revise the summary accordingly.

```
You have to summarize the below
function.
Then critique your summary for
correctness and completeness.
Refine, if necessary.

Function:
{code}
```

5) Expert Prompting: Here, the model is prompted to behave as a seasoned software engineer explaining the function to a junior developer. This format emphasizes readability, completeness, and instructional clarity.

```
Imagine you are a software engineer
with high level of experience.
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 carry out our experiments using datasets provided by the CodeXGLUE benchmark, focusing on three programming languages: Python, JavaScript, and Java. Each dataset comprises matched pairs of source code fragments and corresponding natural language descriptions (e.g., docstrings), which serve as ground-truth references for model evaluation.

Dataset Overview:

- **Python:** 251,820 code-summary pairs consisting of utility routines, algorithms, and class methods sourced from open-source projects.
- **JavaScript:** 58,025 samples largely centered on user interface utilities, DOM manipulation, and asynchronous operations.
- **Java:** 164,923 entries containing method definitions, constructors, and object-oriented programming constructs.

These datasets support our empirical evaluation, where we assess how different prompting formats influence model output across languages and model configurations.

Data Preprocessing:

- **Deduplication:** We removed identical and near-duplicate functions using hash-based fingerprinting to retain a varied dataset.
- **Format Standardization:** We ensured uniformity in indentation, spacing, and comment formatting to minimize inconsistencies in prompt structure.
- **Stratified Sampling:** For experimental testing, five code examples per language were randomly selected, ensuring balanced coverage across all five prompting types and both LLMs.

While CodeXGLUE offers a strong baseline for evaluating summarization tasks, its code snippets are generally short and self-contained. As such, results may not extend directly to large-scale or multi-file codebases that demand a deeper contextual understanding.

V. EXPERIMENTS AND EVALUATION

A. Evaluation Metrics

To measure the quality of the generated summaries, we apply four established metrics: BLEU, METEOR, ROUGE-L, and BERTScore. These metrics jointly assess both the textual fidelity and semantic similarity of the model outputs.

BLEU (Bilingual Evaluation Understudy:) Originally created for evaluating machine translation, BLEU checks how many overlapping n-grams exist between the generated and reference texts. While useful for precision-based scoring, it can sometimes penalize valid paraphrases.

METEOR (Metric for Evaluation of Translation with Explicit ORdering:) Designed to be more forgiving than BLEU, METEOR considers stemming, synonym matching, and word order. It provides more nuanced alignments and better accommodates linguistic variation.

ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation – Longest Common Subsequence:) This metric identifies the longest matching sequence of words between the reference and generated texts, placing emphasis on recall. It is particularly helpful for estimating the completeness and fluency of summaries.

BERTScore: Unlike the other metrics, BERTScore uses contextual embeddings from pretrained BERT models to compute semantic similarity. It is especially effective at detecting when the generated output conveys the same meaning as the reference, even if different words are used.

By using this diverse set of metrics, we obtain a comprehensive evaluation that reflects both syntactic precision and semantic adequacy. All evaluations were performed using publicly available tools, and the final scores were averaged across the test samples for every combination of model, language, and prompting strategy.

VI. RESULTS AND DISCUSSION

A. Overall Model Performance

Table I presents the comparative evaluation of the DeepSeek and LLaMA language models across five prompting strategies and three programming languages—Python, JavaScript, and Java. Overall, LLaMA consistently outperformed DeepSeek in terms of both text similarity (e.g., BLEU, ROUGE-L) and semantic similarity (e.g., BERTScore), particularly under few-shot and chain-of-thought prompting conditions.

Python: For Python, LLaMA consistently outperformed DeepSeek across all prompting strategies and evaluation metrics. Notably, LLaMA's Few-shot prompting yielded the highest scores (BLEU: 0.05134, METEOR: 0.2763, ROUGE-L: 0.26704, BERTScore: 0.62754), demonstrating its superior ability to learn from in-context examples. In contrast, DeepSeek's highest Python performance was under Zero-shot prompting, but its BLEU (0.03742) and BERTScore (0.5845) still lagged behind LLaMA. The performance gap suggests that LLaMA generalizes better in Python summarization tasks, especially when guided with example pairs.

JavaScript: JavaScript proved to be more challenging for both models, with generally lower scores. However, LLaMA again demonstrated stronger semantic alignment, achieving its best BERTScore (0.4642) and METEOR (0.15328) under Chain-of-thought prompting. BLEU scores remained low for both models, but LLaMA showed a broader range of competent prompting styles compared to DeepSeek, which had flat BLEU scores across all strategies (often 0). This indicates that LLaMA is better suited for handling JavaScript's asynchronous and dynamic constructs when supplemented with reasoning-based prompts.

Java: In Java, LLaMA showed notable strength in semantic metrics, especially under Chain-of-thought (BERTScore: 0.53676) and Few-shot (BERTScore: 0.48572) prompting. BLEU scores remained low or zero, similar to DeepSeek. DeepSeek achieved its highest Java BERTScore (0.46028) under Zero-shot prompting, but quickly tapered off under more structured prompts. LLaMA's more consistent performance across strategies in Java suggests stronger generalization and contextual understanding, whereas DeepSeek displayed more variance and lower semantic correlation.

Prompting Strategy Comparison: Across both models, Few-shot prompting emerged as the most effective overall, particularly for LLaMA. It yielded the highest combination of BLEU and BERTScore across all languages. Chain-of-thought and Critique strategies offered moderate gains—especially for LLaMA—whereas Zero-shot prompting performed weakest across models and languages (especially for LLaMA and JavaScript). Interestingly, DeepSeek benefited more from Zero-shot prompting in Python and Java, while LLaMA benefited more from Few-shot and Chain-of-thought prompting across all languages, underscoring LLaMA's flexibility in using contextual and reasoning cues.

B. Limitations

While the study provides meaningful insights into how large language models (LLMs) handle code summarization, there are several factors that may limit the generalizability of the findings:

- **Inaccurate Summary Generation:** At times, particularly when using zero-shot or expert-style prompting, models produced summaries that misrepresented the actual function of the code. This hallucination effect was more frequent for ambiguous or sparsely commented code snippets.
- **Dataset Limitations:** The primary datasets employed—CodeXGLUE and CodeSearchNet—consist mainly of short, standalone functions. Such examples may not adequately represent the complexity of production-level codebases, which typically span multiple files and contain intricate module interactions.
- **Hardware Constraints:** Due to limited computational resources, including available memory and processing power, the study was restricted to testing three programming languages and two mid-sized LLMs. Larger models, such as DeepSeek-Coder R1:14B, LLaMA 3.3:70B, or open alternatives like StarCoder, were excluded for practical reasons.
- **Prompt Sensitivity:** The quality of generated summaries often varied with subtle alterations in prompt phrasing or layout. This inconsistency suggests that some models are highly sensitive to input structure, making reproducibility more difficult.
- **Limited Evaluation Scope:** The study employed four common metrics—BLEU, ROUGE-L, METEOR, and BERTScore—for performance evaluation. Although effective in measuring lexical and semantic overlap, these metrics do not assess how helpful or functionally accurate the summaries are in real-world programming tasks. Manual review or task-specific evaluations (e.g., debugging support) would offer deeper insights.

C. Performance Review of DeepSeek:

To examine how the DeepSeek language model performs in code summarization, we analyzed its outputs under five prompting frameworks—Zero-shot, Few-shot, Chain-of-thought, Critique, and Expert—across three languages: Python, JavaScript, and Java. Each result was evaluated using a combination of BLEU, METEOR, ROUGE-L, and BERTScore metrics. These scores reflect different facets of similarity, from lexical overlap to semantic alignment, between the generated summary and the human-authored reference.

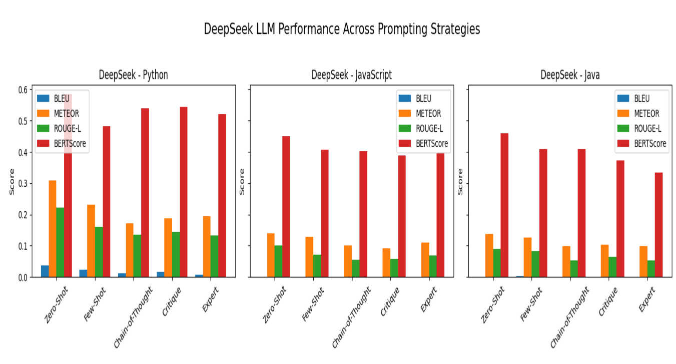

Fig. 1: Comparison in performance of DeepSeek LLM by employing different prompting strategies across Java, Python, JavaScript, using ROUGE-L, METEOR, BLEU, and BERTScore.

TABLE I: Evaluation Results of LLM Models with Various Prompting Techniques Across Programming Languages

Sr No.	LLM Models	Prompting Techniques	Python				JavaScript				Java			
			Text Similarity			Semantic Similarity	Text Similarity			Semantic Similarity	Text Similarity			Semantic Similarity
			BLEU	METEOR	ROUGE-L	BERT Score	BLEU	METEOR	ROUGE-L	BERT Score	BLEU	METEOR	ROUGE-L	BERT Score
1	DeepSeek	Zero-Shot	**0.03742**	**0.30776**	**0.22292**	**0.5845**	0	**0.13994**	**0.10032**	**0.45038**	0	**0.13812**	**0.08936**	**0.46028**
		Few-Shot	0.02456	0.23134	0.1594	0.48246	0	0.12938	0.07252	0.40588	**0.00212**	0.12526	0.08252	0.40924
		Chain-of-Thought	0.01234	0.17274	0.13532	0.53844	0	0.10182	0.05478	0.4022	0.00122	0.09974	0.05314	0.40922
		Critique	0.01628	0.1889	0.14416	0.544	0	0.09126	0.05756	0.38916	0.00174	0.10364	0.06466	0.37172
		Expert	0.0067	0.19386	0.13236	0.52194	0	0.11098	0.06916	0.3966	0.0016	0.0988	0.05358	0.33384
2	LLaMA	Zero-Shot	**0.05472**	0.20618	0.23456	0.52134	**0.0098**	0.11432	**0.14728**	0.40802	0	0.0974	**0.2162**	0.1991
		Few-Shot	0.05134	0.2763	**0.26704**	**0.62754**	0.0082	0.15042	0.13396	0.45362	0	**0.19858**	0.14386	0.48572
		Chain-of-Thought	0.04096	**0.30152**	0.2323	0.56434	0	0.15328	0.13016	0.4642	0	0.18396	0.10194	**0.53676**
		Critique	0.0203	0.28626	0.21006	**0.64996**	0.00368	0.14096	0.09616	0.4516	0	0.15262	0.1121	0.48234
		Expert	0.02714	0.25448	0.22626	0.5798	0	**0.1709**	0.11204	**0.50616**	0	0.16288	0.11174	0.4467

The results indicate that Python achieved the highest scores across nearly all metrics and prompting styles. Among the prompting techniques, the Zero-shot strategy achieved the highest BLEU score (0.03742) and ROUGE-L score (0.22292) in Python, while METEOR peaked at 0.30776, and BERTScore reached 0.5845, showcasing the model's relative strength in handling Python code with minimal guidance. Notably, BERTScore remained high across all prompting methods, indicating strong semantic alignment with reference summaries.

In contrast, JavaScript and Java presented greater challenges for DeepSeek. For both languages, BLEU scores were consistently close to zero, with slight improvements under Few-shot and Critique strategies. For instance, in JavaScript, METEOR reached 0.13994 and ROUGE-L 0.10032 in the Zero-shot setting, while BERTScore remained modest at 0.45038. Similarly, Java displayed a METEOR of 0.13812 and ROUGE-L of 0.08936 under Zero-shot prompting, with BERTScore slightly better at 0.46028.

Interestingly, while Few-shot prompting did not lead to the highest BLEU scores, it produced relatively balanced outcomes across all metrics and languages, suggesting it helps DeepSeek generate more context-aware and semantically aligned summaries. Critique prompting consistently yielded stable BERTScores across languages, indicating that structured feedback may help refine the summarization process, especially in semantically complex cases.

Conversely, Chain-of-thought prompting, although beneficial in some LLMs, did not significantly enhance DeepSeek's performance. BLEU and ROUGE-L scores remained low, particularly for JavaScript and Java. Similarly, Expert prompting, which mimics human-like explanations or annotations, showed moderate gains in semantic metrics like BERTScore (up to 0.52194 in Python) but remained limited in text-overlap metrics like BLEU and ROUGE-L.

Overall, the results suggest that **DeepSeek performs best on Python code**, likely due to its syntactic clarity and abundant training data. Prompting strategies like **Few-shot and Critique generally improve semantic quality**, while **Zero-shot and Chain-of-thought promptings appear less effective**1. These insights emphasize the necessity of both model selection and prompt design when deploying LLMs for source code summarization tasks.

D. Model Performance Comparison: LLaMA

The LLaMA model's performance was evaluated across the same set of five prompting strategies—Zero-shot, Few-shot, Chain-of-thought, Critique, and Expert—and three programming languages: Python, JavaScript, and Java. Its summarization capabilities were assessed using standard metrics: BLEU, METEOR, ROUGE-L, and BERTScore, covering both surface-level and semantic similarity with reference summaries.

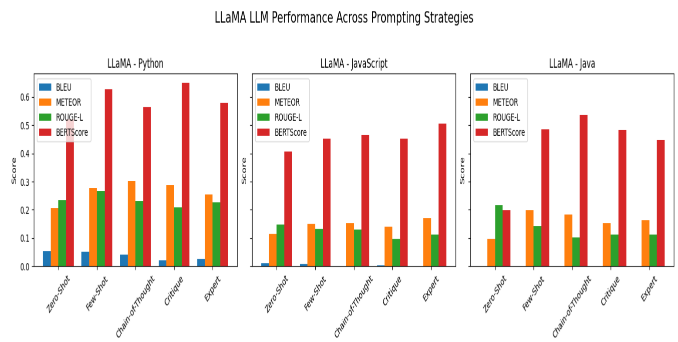

Fig. 2: Comparison in performance of LLaMA LLM by employing different prompting strategies across Java, Python, JavaScript, using ROUGE-L, METEOR, BLEU, and BERTScore.

LLaMA showed comparatively stronger performance on Python, with the Few-shot prompting strategy achieving the highest scores across all four metrics—BLEU (0.05134), METEOR (0.2763), ROUGE-L (0.26704), and BERTScore (0.62754). This trend highlights Few-shot's effectiveness in guiding the model via example summaries, helping it better align both lexically and semantically with expert-generated references.

Interestingly, Chain-of-thought prompting also delivered competitive results in Python with METEOR (0.30152) and ROUGE-L (0.2323), suggesting that the reasoning steps incorporated in the prompts benefited LLaMA more than DeepSeek. However, BLEU remained lower (0.04096), indicating possible divergence in exact word overlap despite semantic improvements.

For JavaScript, performance improved relative to DeepSeek, with BLEU peaking at 0.0098 (Zero-shot) and BERTScore improving under Few-shot (0.45362) and Chain-of-thought (0.4642) strategies. These results underscore LLaMA's stronger semantic understanding even in languages with more asynchronous and dynamic syntax.

In Java, LLaMA also demonstrated stable behavior with semantic scores—BERTScore reaching 0.53676 under Chain-of-thought, and 0.48572 under Few-shot. While BLEU remained near zero, the consistent rise in BERTScore across prompting styles implies the model's ability to capture high-level meaning even if not precisely matching surface n-grams.

Critique prompting was somewhat effective in all languages, offering decent METEOR and BERTScore values, particularly in Python and Java. Expert prompting, while not the top performer, remained competitive with BERTScore values like 0.5798 (Python) and 0.50616 (JavaScript), suggesting that structured guidance helps, though not as strongly as in-context examples.

Overall, **LLaMA exhibits robust summarization performance**, particularly **with Python code and Few-shot prompts**2. Compared to DeepSeek, it displays **better generalization across JavaScript and Java**, as evidenced by its higher semantic scores in these languages. These findings affirm LLaMA's capability to leverage prompt engineering effectively and produce summaries that are not only lexically coherent but also semantically aligned with expert interpretations.

VII. CONCLUSION

This study provides a comprehensive evaluation of LLMs for source code summarization, identifying their capabilities and limitations. The insights gained will be valuable for both academia and industry, fostering further research and practical advancements in AI-driven software development tools.

The conclusion of this research, it is found that prompt engineering and its efficiency has a substantial role in improving the code summarization quality. The findings showcase how high-quality summaries are being generated by the chosen prompting strategies.

VIII. LIMITATIONS AND FUTURE DIRECTIONS

- **Domain-Specific Source Code Summarization:**
  Explanation: Our current work evaluates general-purpose code summarization across multiple programming languages using mainstream LLMs. However, in real-world software engineering, many projects operate within specific domains—such as Web Development, AI/ML, Embedded Systems, Data Science, Cybersecurity, or Finance—each with unique libraries, frameworks, and terminologies.
  Future Goal: To develop and evaluate LLM-based summarization systems fine-tuned or adapted for domain-specific codebases by curating the datasets. For example:
  - Summarizing code in financial applications (e.g., banking software in Java).
  - Generating comments for machine learning scripts in Python using domain-aware terminology (like "gradient descent" or "tensor operations").
- **Reverse Code Summarization (Natural Language to Code):**
  Explanation: Although the current research emphasizes transforming source code into natural language summaries (code $\rightarrow$ text), an equally important and complementary task is reverse code summarization, where natural language descriptions are used to synthesize source code (text $\rightarrow$ code). This direction holds considerable promise for advancing tools that support:
  - Natural language-based programming interfaces, where developers can describe functionality in plain language and receive corresponding code implementations.
  - Intelligent code completion and generation systems, such as GitHub Copilot and Amazon CodeWhisperer, which rely on robust text-to-code capabilities to assist developers in real-time.

  Future Objective: A planned extension of this work is to evaluate the effectiveness of general-purpose LLMs—specifically DeepSeek and LLaMA—in generating correct, syntactically valid, and functionally accurate code from natural language inputs. This would involve assessing the models on datasets designed for code synthesis and benchmarking them across multiple programming languages.
- **Summarization of Long Functions, Classes, or Modules:**
  Overview: The current evaluation primarily focuses on short, isolated code snippets. However, real-world software development often involves understanding and documenting much larger code blocks, such as entire functions, class definitions, or modules.
  Why it matters: Summarizing longer code units poses additional challenges due to their increased complexity, deeper nesting, and richer contextual dependencies. Tackling this would test a model's ability to:
  - Handle long-range dependencies within the code.
  - Generate coherent and comprehensive summaries that cover the broader intent and structure.
  - Align more closely with practical developer workflows, such as documenting APIs, refactoring legacy systems, or generating class-level documentation.

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

Presented below is a sample **Python** function that was part of our evaluation set. Following the function, we include the ground truth (human-written) summary and the summaries generated using various prompting techniques with the **DeepSeek** model.

Input Code Snippet:

Listing 1: Example Function Used for Summarization

```python
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 internal method is triggered after an element is appended to another. It ensures the correct document is assigned and may perform additional checks. If conditions warrant, it raises exceptions to block invalid additions. Primarily intended for backend functionality.

LLM-Generated Summaries (Deepseek):

1) Zero-Shot:

Alright, I need to create a short comment in one sentence for the function defined. Let me first understand what this function does.
The docstring explains that the method is called after an element is added and does some checks. It can raise exceptions if necessary. By default, it ensures the right document is associated. For internal use only.
Looking at the code:
1. The first check: If self.doc is false and parent’s doc is true, set the child’s doc to parent’s. 2. Then, if self.doc is true and has deepvalidation, call that method.
So, the function ensures each element is properly linked with its document by default. It raises exceptions if necessary for internal use. The comment should reflect this behavior concisely in one sentence.
Maybe something like ”The function ensures correct document association between elements.” That covers both default behavior and potential internal use aspects. ¡/think¿
”The function ensures correct document association between elements.”

2) Few-Shot:

Okay, I need to summarize the given Python function. Let me read through it carefully.
The function is called postappend, and it’s using a class object with a ’self’ attribute. The docstring explains that this method is called after elements are added to another and does some checks. It can do extra checks if needed and raises exceptions if something goes wrong. By default, it ensures the right document is associated.
Looking at the code inside, there’s an if statement checking if ’doc’ is true but not in the parent. If that’s the case, it sets the document to the parent’s doc. Then another if checks if ’self.doc’ and ’self.doc.deepvalidation’ are both true, so it calls self.deepvalidation().
I think this function is part of a class hierarchy, maybe for adding elements where each element has its own document. The first check ensures that any new element gets the correct parent’s doc if needed. Then, if deepvalidation is needed on this element or its children, it runs that check.

The user provided an example where add(a,b) returns their sum. So in postappend, they might be doing similar validations after adding elements to ensure everything is set up correctly.
I should make sure I include all the checks and exceptions mentioned. It's mostly for internal use, so maybe not public or part of a standard library.
¡/think¿
The function ‘postappend‘ ensures that each element is properly associated with its document by calling the necessary validations. Here's how it works:
1. **Check Parent Document**: If the current element (‘self‘) doesn't have a document and neither does its parent, set the current element's doc to the parent's doc. 2. **Check DeepValidation on Current Element or Children**: If both ‘self.doc‘ and ‘deepvalidation()‘ are true, run the deep validation.
This ensures that validations are done before adding new elements and associated children, preventing errors by raising exceptions when issues arise.

3) Chain-of-Thought:

Alright, I'm trying to understand this Python function. Let me break it down step by step.
First, the function is called postappend, which suggests that something happens after an element is added or processed. It's a method of some class since 'self' is used.
The docstring gives a good overview. The method checks if 'doc' and 'parent.doc' are both not set, and if so, it sets the document to the parent's document. Then, there's a check for deepvalidation, which calls self.deepvalidation() after that.
I'm wondering why the function is called postappend. Maybe because it happens after an element has been added or processed, hence "after". Also, I notice that 'self' is being referenced again, so each method call should have access to the instance and its attributes.
Wait, in Python, methods are defined within a class using 'class Object: def method(self): ...'. So in this function, self will refer to an instance of the class it's defined in. But in the code, I see that 'self.doc' is accessed, which might not be correct because the instance variable 'doc' hasn't been set yet.
Hmm, maybe the initial part sets some variables before calling deepvalidation? Like:
if not self.doc and self.parent.doc: self.setdocument(self.parent.doc)
So perhaps after setting this document, it calls deepvalidation. But looking at the code, the line is just 'self.deepvalidation()', without any exceptions or conditions.
I'm thinking that maybe there are some error handling missing here. What if deepvalidation raises an exception? It's not caught anywhere. That could be a problem because it might crash the program unexpectedly.
Another thing I notice is that after setting the document, it doesn't call another method called 'deepvalidation()'. Maybe that method should have been called before setting the document?
Also, I'm trying to remember how exceptions are handled in Python. If something raises an error during deepvalidation, the function might not handle it, leading to issues.
I think the main points are:
1. The docstring gives a clear explanation of what the function does. 2. There's missing exception handling for self.deepvalidation() which could cause errors. 3. The order of operations seems off; perhaps setting the document should happen before deepvalidation is called.
To summarize, I need to understand that this function sets up a document after an element is processed and calls some internal validation method. It might be missing steps or not handling exceptions properly.
¡/think¿
Based on the code and reasoning:
1. **Function Overview**: - The ‘postappend‘ method ensures the right document is associated after adding an element, primarily for internal use.
2. **Key Features**: - Checks if the current instance (‘self‘) is not already linked to a document. - If necessary, sets the document to the parent's document.
3. **Potential Issues**: - Missing exception handling for ‘deepvalidation()‘ call, which may lead to runtime errors. - Unclear order of operations: The code sets up the document and calls ‘deepvalidation‘ without specifying if it should be executed before or after.
4. **Recommendations**: - Add error handling around ‘self.deepvalidation()‘ to catch exceptions and prevent crashes. - Ensure that ‘deepvalidation‘ is called before setting up the document for a more robust flow.

4) Critique:

Okay, so I need to summarize this function called postappend. Let me read through the code again carefully.
The function starts with a docstring explaining what it does. It says it's called after an element is added and does some checks. By default, it makes sure the right document is associated. This method is mostly for internal use.
Looking at the code:
1. The first line inside the method is checking if ‘self.doc‘ is False and ‘self.parent.doc‘ is also False. If both are false, then it sets ‘self.doc‘ to ‘self.parent.doc‘.
2. Then there's a condition where if ‘self.doc‘ exists

(True) and ‘self.doc.deepvalidation‘ is True, it calls ‘self.deepvalidation()‘.
Hmm, I notice that the first line isn’t properly indented after the docstring. It should be under both conditions, not just after setting ‘self.doc‘. Also, perhaps the function should handle cases where a child might need to set its own document without relying on the parent. That would make it more flexible and reduce the risk of conflicts.
So, I think the summary should mention that this method ensures the correct doc is set when an element is added, handles internal checks for validation, and allows children to manage their own docs. It’s important to note that the parent’s document defaults if none exists, but each child can override it, making the function more robust.
Wait, in the original code, after checking ‘self.doc‘ and ‘self.parent.doc‘, they set ‘self.doc‘. But what about when a child doesn’t have an associated doc yet? They should set their own. So the summary needs to reflect that flexibility without missing any key points.
Also, I should make sure that the summary is concise but covers all main points: handling validation checks, defaulting to parent’s document, allowing children to manage their own docs, and being internal for internal use. ¡/think¿
Summary of ‘postappend‘ Method:
- The ‘postappend‘ method ensures that when an element is added, it gets associated with the correct document. - If the current element (‘self‘) lacks a document, it inherits the document from its parent. - If ‘deepvalidation‘ is enabled on the document, it performs validation via ‘self.deepvalidation()‘. - The method allows flexibility: child elements can manage or override their own documents after association. - It is primarily used for internal validation and setup logic after element insertion.

5) **Expert:**

Okay, I need to summarize this function explanation. The user provided the function’s code and wants it explained in a way that a junior developer would understand. Let me break down what each part does.
First, there’s an ‘if‘ condition checking if both ‘self.doc‘ and ‘self.parent.doc‘ are not ‘None‘. If they’re both set, it calls ‘self.setdocument(self.parent.doc)‘. This means the current node is being associated with its parent’s document. That’s important because it ensures consistency — a node shouldn’t exist without being linked to a valid document context.
Next, there’s another ‘if‘ condition: if both ‘self.doc‘ and ‘self.doc.deepvalidation‘ are ‘True‘, it calls ‘self.deepvalidation()‘. Deep validation is likely used to enforce internal constraints — things like content formatting or structure. If the content doesn’t pass validation, the function will raise an exception to prevent invalid nodes from being added.
It’s worth noting that this function is intended for internal use only. It’s not designed to be used directly by developers, but rather to handle internal consistency and validation when nodes are added to a structure.
I should structure this step-by-step: first, it ensures the node has a document by inheriting it from the parent if necessary. Then, it performs validation if required. The main idea is to maintain document integrity and enforce validation rules.
¡/think¿
Summary of ‘postappend‘ Method:
- Ensures that a node is linked to its parent’s document using ‘self.setdocument(self.parent.doc)‘, maintaining consistency in document structure. - If ‘deepvalidation‘ is enabled, it calls ‘self.deepvalidation()‘ to enforce internal constraints. - Raises an exception if validation fails, preventing the node from being added incorrectly. - This method is for internal use only, supporting safe and structured updates when new nodes are appended.

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 **Llama** 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 (Llama):

1) **Zero-Shot:**

This function appears to be intended to check if a given value is within a specified integer range.

2) **Few-Shot:**

This function determines whether a given input value falls within a specified numeric range. It works by

first converting the input value to an integer, then performing the following checks:
1. **String Match Without Leading Zeros**: It verifies that converting the integer back to a string exactly matches the original input value after leading zeros are removed. This ensures that inputs like ‘"007"‘ are rejected, while ‘"7"‘ is accepted.
2. **Lower Bound Check**: It confirms that the integer is greater than or equal to the minimum value of the allowed range.
3. **Upper Bound Check**: It confirms that the integer is less than or equal to the maximum value of the allowed range.
If all three checks pass, the function returns ‘true‘, indicating the value is valid and within the range. Otherwise, it returns ‘false‘.

3) Chain-of-Thought:

Let's break down the function step by step:
1. Function Purpose: The ‘inRange‘ function checks if a given value falls within a specified range, defined by two minimum and maximum values.
2. Inputs: - value: The number to be checked. - min: The lower bound of the range (inclusive). - max: The upper bound of the range (inclusive).
3. Outputs: The function returns a boolean value (true or false) indicating whether the input value is within the specified range.
4. Function Summary: The ‘inRange‘ function checks if a given number is an integer and falls between two specific minimum and maximum values, inclusive. If the value meets these conditions, it returns true; otherwise, it returns false.

4) Critique:

The ‘inRange‘ function checks if a given value is within a specified range defined by two minimum and maximum integers. It does this by first converting the input value to an integer, removing any leading zeros from the original string representation of the integer, and then comparing it with the original string to ensure they are the same (i.e., there were no leading zeros in the original value). The function returns ‘true‘ if the converted integer is within the specified range.
Critique:
The summary is concise but could be improved. Here's a refined version that adds more detail and clarity:
Summary:
The ‘inRange‘ function determines whether an input value falls within a specific numeric range defined by two integers, ‘min‘ and ‘max‘. It performs this check in the following steps: 1. Converts the input value to an integer using ‘parseInt‘. 2. Removes leading zeros from the original string representation of the integer. 3. Compares the converted integer with its original string representation to ensure they are identical (i.e., there were no leading zeros). 4. Verifies that the converted integer is within the specified range (‘min‘ and ‘max‘) defined by integers.
Refinement:
This refined summary includes more detail about the function's behavior, making it easier for readers to understand its purpose and functionality.

5) Expert:

Here's a concise explanation of the function:
This 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()‘. - Then, it performs three conditions: - ‘value‘ must be equal to its numeric value (i.e., not a string representation of a number with leading zeros). - The integer value must be greater than or equal to ‘min‘. - The integer value must be less than or equal to ‘max‘.
If all these conditions are met, the function returns ‘true‘, indicating that the ‘value‘ is within the range. Otherwise, it returns ‘false‘.

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 **Deepseek** 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 (Deepseek):

1) Zero-Shot:

Alright, I need to respond to this user's query about generating a short comment for their function. The function is public ¡T¿ T getNodeMetaData(Object key) ... , and they want one sentence in the response. First, I'll look at what the function does. It's part of some object handling or maybe graph traversal code. The parameter 'key' is used to get metadata from a map called metaDataMap. If the map is null, it returns null; otherwise, it retrieves and returns the corresponding value as type T.
So, the main points are: the function uses a key to look up metadata, it checks if the map exists before accessing it, and handles the case where the map might be null.

Now, how can I condense this into one concise comment? The user probably wants a quick understanding of what the function does without looking at the code. I should focus on the key parameter's role and the return behavior based on metaDataMap. I think mentioning that it returns T or null if metaDataMap is null would cover both possibilities. It clearly shows how the key is used to fetch data, so the comment should reflect that functionality concisely.
The function retrieves metadata for a given key using ‘metaDataMap‘, returning ‘T‘ or ‘null‘ if the map is null.

2) Few-Shot:

Alright, let's see. The user provided two examples and a new function they want summarized. I need to figure out what the function is doing based on the examples.
First example is an add function that takes two numbers and returns their sum. That makes sense—adding two values together. So in the case of getNodeMetaData, if metaDataMap is null, it returns null; otherwise, it retrieves the value from metaDataMap using the key.
The getNodeMetaData function seems to check if metaDataMap is null; if so, it returns null. Otherwise, it retrieves the entry for the given key and returns it as an object of type T. That makes sense in the context where metaDataMap contains various types of data that can be returned by this method.
I should explain that the function behaves like a default value provider: when the map isn't available (null), it falls back to the retrieved entry, preserving the return type as per the key's presence.
The provided function ‘getNodeMetaData‘ appears to act as a simple lookup mechanism. Here's how it works:
1. **Null Check**: The function first checks if ‘metaDataMap‘ is null. If it is, the function returns ‘(T) null‘. 2. **Default Case**: If ‘metaDataMap‘ is not null, the function retrieves the value associated with the given key from ‘metaDataMap‘. 3. **Return Type Preservation**: The retrieved value (if present in ‘metaDataMap‘) will be returned as an object of type ‘T‘, ensuring that the return type remains consistent.
In summary, ‘getNodeMetaData‘ acts as a fallback mechanism, returning a default value when no mapping is found, while preserving the expected return type based on the presence of data in ‘metaDataMap‘.

3) Chain-of-Thought:

1. **Step 1: Understand what the function does** - This function is called ‘getNodeMetaData‘ in Java. - It takes one parameter, which is an Object called ‘key‘. - The function returns a value that depends on the return type ‘T‘.
2. **Step 2: Identify the inputs and outputs** - **Input**: ‘key‘ is of type Object. - This means the function accepts any object as its input. - **Output**: The returned value's type depends on how ‘metaDataMap.get(key)‘ behaves. For example, if ‘metaDataMap‘ returns a String for every key, then the output will be a String.
3. **Step 3: Summarize it clearly** - This function retrieves metadata associated with a given key. - If no metadata is stored (i.e., ‘metaDataMap‘ is null), the function returns null. - Otherwise, it retrieves and returns the metadata value corresponding to the provided key.
This explanation captures the essential functionality of the code without unnecessary details. The user can use this understanding in context where they need to integrate or extend this logic elsewhere.

4) Critique:

Okay, so the user provided a function and asked me to summarize it and critique my summary. Let's break this down.
First, I'll read through the function they gave: public ¡T¿ T getNodeMetaData(Object key). The first line checks if metaDataMap is null and returns null in that case. If not, it gets the value from metaDataMap using the key and casts it to T. So, it's a method that retrieves metadata for a given object key.
Now, I need to summarize this function properly. My summary should be clear and concise but also capture all necessary details without being too wordy. Let me think about each part:
1. The function is public ¡T¿ so it can return any type T. 2. It takes an Object key as input. 3. There's a null check on metaDataMap, which if true, returns null immediately. 4. Otherwise, it fetches the metadata using get with the key and casts to T.
I should make sure all these points are included in my summary without adding extra information that isn't part of the function. Maybe I can combine some parts for clarity. For example, combining the if statement into one sentence or explaining the flow better.
Let me also consider if there's anything missing. Does it handle cases where key is null? Probably not specified here, but maybe in a real scenario, they'd handle that separately. So, perhaps mention returning null if key is null as well, though the current code doesn't do that yet.
Another point: what type T represents? If it's an array or some generic type, clarifying would help. But without knowing, I can leave it generic for now.
So, putting it all together: a function that returns a value of type T, takes an Object key, checks if meta-

DataMap is null and returns null if so. Otherwise, retrieves the metadata using get with key and casts to T.
The function ‘getNodeMetaData‘ is designed to retrieve metadata for a given object key. Here’s a clear summary of its functionality:
1. **Function Type**: It returns an object of type ‘T‘. 2. **Input Parameter**: It takes an ‘Object‘ parameter named ‘key‘. 3. **Null Check**: If the ‘metaDataMap‘ is null, the function immediately returns a ‘null‘ value. 4. **Metadata Retrieval**: If ‘metaDataMap‘ is not null, the function retrieves the metadata for the provided ‘key‘ using the ‘get‘ method and casts it to type ‘T‘.
In summary, this function checks if metadata mapping data exists, retrieves or returns it based on the key, and handles a null check before execution.

5) Expert:

Okay, so I’m trying to figure out how to explain this function as if I’m talking to a junior developer. Let me read through the code first.
The function is called getNodeMetaData and takes an Object key as input. It returns something of type T, which I assume is a generic type T because it’s used with other types like null or whatever else comes next.
Hmm, so if metaDataMap is null, it just returns null. That makes sense for early termination. Otherwise, it calls get on the metaDataMap with the key and casts the result to T. Wait, I wonder what kind of objects are in metaDataMap and how that affects T. In Java or JavaScript, using a null-safe approach with Objects can prevent NullPointerExceptions when someone doesn’t have the key in the map. The function handles that by returning null instead of throwing an error. That’s a good practice for code maintainability.
I think I need to explain what each line does clearly. Maybe start with the overall purpose: getting metadata for a given key from some map, handling missing keys gracefully. Also, mention how the function uses Object-Oriented Programming principles like encapsulation and early termination.
Wait, is metaDataMap of type Object? Because in Java or JavaScript, if it’s null, get would return null. In other languages, sometimes you have to handle nulls differently. But since I’m assuming a similar structure here, that should be fine.
I wonder if the function is called with any specific types for key and T. Maybe someone could pass in a custom type for key or expect T to be something else. So it’s important to note that key must match what metaDataMap uses as keys, otherwise, you get a null in T, which might not behave as expected.
Also, the return type T can represent various things like properties, methods, etc., depending on the context where this function is used. But since it’s generic, it’s flexible enough to handle different scenarios.
I should also mention how this helps with code structure and readability. It keeps related operations in one place, reducing redundancy and making the codebase easier to manage. Especially when dealing with multiple similar functions that extract metadata from a map.
Oh, right! I remember that using Object-Oriented Design principles can make the code more maintainable. If other parts of the system depend on this function’s behavior, they won’t have to handle it separately, which is great for scalability and future-proofing.
So putting it all together: explain the function, its purpose, how it handles missing keys with null return, generic return type, object-oriented approach for better structure. That should give a clear picture of why this code is written.
To provide a clear and concise explanation of the ‘getNodeMetaData‘ function, here’s an organized summary:
1. **Purpose**: This function retrieves metadata for a given key from a map called ‘metaDataMap‘. It handles missing keys gracefully by returning ‘null‘.
2. **Null Handling**: If ‘metaDataMap‘ is null, it returns ‘null‘, preventing a potential NPE.
3. **Generic Return Type**: The result is returned as an Object type (T), allowing flexibility in the metadata representation across various uses.
4. **Early Termination**: By checking if ‘metaDataMap‘ is null upfront, the function ensures quick termination when the key is not found, enhancing efficiency and readability.
5. **Object-Oriented Design**: Using a generic return type promotes code structure and maintainability, reducing redundancy and future refactoring efforts.
This approach emphasizes code clarity, maintainability, and scalability, making it ideal for managing related operations in a structured manner.

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