Reliable Inline Code Documentation with LLMs: Fine-Grained Evaluation of Comment Quality and Coverage

Rohan Patil
Western Digital, India
rohan.patil1@wdc.com

Gaurav Tirodkar
Western Digital, India
gaurav.tirodkar@wdc.com

Shubham Gatfane
Western Digital, India
shubham.gatfane@wdc.com

Abstract

Code documentation plays a vital role in enhancing collaboration, maintainability, and comprehension throughout the software development lifecycle. This becomes especially critical in legacy codebases, where missing or outdated comments hinder effective debugging and onboarding. Among documentation types, inline comments are particularly valuable for conveying program logic and supporting code reuse. With the growing capabilities of large language models (LLMs), their application to tasks such as code understanding and summarization has gained significant attention in the NLP community. However, the specific task of generating high-quality inline code comments using LLMs remains relatively under-explored. In this work, we conduct a systematic evaluation of several state-of-the-art LLMs to assess their effectiveness in producing meaningful and context-aware inline documentation. To this end, we curate a dataset of well-documented code snippets and propose a fine-grained evaluation framework that assesses both the quality and sufficiency of generated comments at the statement level. We further investigate the impact of prompting strategies and offer a comparative analysis across a range of models, including large foundational LLMs to smaller, code-specialized variants, within the domain of inline code documentation. Our findings offer actionable insights that can guide the development of effective and scalable systems for automated inline code documentation.

1 Introduction

Good quality code documentation is essential for the sustainability, readability, and maintenance of software projects. It facilitates onboarding, reduces the learning curve, and accelerates time-to-market. Inline and block comments are particularly important as they summarize code sections, explain assumptions, and describe control flow, thereby improving interpretation of software modules. However, writing rich, developer-level documentation requires significant time and effort, often reducing developer productivity. Xia et al. (2018) in their study show that developers spend nearly 59% of their time on program comprehension during software development, underscoring the need for automated tools to improve efficiency through high-quality inline comments.

Large Language Models (LLMs) have demonstrated strong performance in code-related tasks, benefiting from training corpora enriched with multilingual programming data. While they show promise in generating summaries and function-level comments, systematic evaluation of their capabilities for producing meaningful inline comments remains limited. Such evaluation must assess not only comment quality but also whether comments are added to the necessary sections of the code without compromising its readability.

In this paper, we investigate the ability of LLMs to generate inline comments using a curated dataset of developer-written code snippets. Starting from The Vault corpus Nguyen et al. (2023), we derive a filtered dataset of functions with inline comments and evaluate multiple LLMs under zero-shot and few-shot prompting. We emphasize balancing comment **quality** and **coverage**, proposing an algorithmic approach that quantifies semantic alignment and sufficiency via an optimal comment-to-code ratio.

We address the following research questions through systematic experimentation:

- **RQ1**: How well do LLMs generate inline comments that align with developer-written standards in terms of semantic quality?
- **RQ2**: Can smaller, code-specialized models match the performance of larger foundational models in inline comment generation?
- **RQ3**: What role do prompting strategies play in enhancing comment quality?

This work contributes to the understanding

Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems, pages 40–54

of inline comment generation through: (1) a language-agnostic evaluation framework that derives $\mathbf{IC_{score}}$, a metric capturing semantic alignment and coverage of block-level comments; (2) a benchmarking study across foundational and code-specialized LLMs using $\mathbf{IC_{score}}$; and (3) an analysis of prompting strategies, comparing zero-shot and few-shot setups to assess their impact on comment quality and guide prompt design for code documentation.

2 Related Work

Several prior studies have investigated the capabilities of NLP models in generating inline code comments. Huang et al. (2023) present an empirical comparison between method-level and inline comments, revealing a notable decline in model performance when generating inline comments. Their findings underscore the inherent difficulty of this task, attributed to the need for fine-grained contextual understanding, and motivate the development of more context-aware and adaptable generation methods.

More recent work has focused on leveraging large language models (LLMs) for code documentation, primarily at the function or module level. Dvivedi et al. (2024) evaluate both proprietary and open-source LLMs across multiple documentation granularities, while Sun et al. (2025b) examine how varying the context window affects the quality of generated documentation. Bappon et al. (2024) specifically target inline comment generation for code snippets from Q&A platforms like Stack Overflow, demonstrating that enriching the input with additional context improves comment quality. However, these studies rely exclusively on human evaluation for assessing the quality of generated comments. With the growing availability of well-documented code in large-scale repositories and community-curated platforms, evaluation settings that include high-quality ground truth are becoming increasingly common. Yet, existing work does not propose automated metrics to assess semantic sufficiency or coverage in such contexts - a gap this work directly addresses.

The evaluation of LLMs for code summarization has also received considerable attention. Studies such as Geng et al. (2024), Szalontai et al. (2024) and Sun et al. (2025a) benchmark models of varying scales, from compact code-specialized models to large foundational LLMs, under different in-context learning setups. These evaluations typically rely on surface-level metrics such as BLEU, ROUGE, or METEOR, or use model-based scoring for contextual relevance. While informative for summarization tasks, these approaches overlook the dual challenge of semantic adequacy and coverage that is central to inline comment generation.

Notably, some of recent analyses have also questioned the reliability of standard metrics. Haldar and Hockenmaier (2024) demonstrate that scores often reflect superficial token overlap rather than genuine semantic understanding, while Song et al. (2024) propose FineSurE, a multi-dimensional framework for evaluating natural language summaries. However, these approaches remain limited to sentence-level abstraction and do not address the unique demands of inline comment generation. Our work fills this gap by introducing an automated metric that jointly captures semantic relevance and coverage, tailored specifically to code block-level comment placement.

3 Method

3.1 Task Definition

Let $x \in \mathcal{X}$ denote a code snippet without inline comments, and let $y \in \mathcal{Y}$ represent the corresponding code with meaningful inline comments inserted at appropriate locations. Let $l \in \mathcal{L}$ be an optional set of few-shot examples, where each example is a pair (x', y') of uncommented and commented code. Let $i \in \mathcal{I}$ denote the natural language instructions in the prompt that guides the conversion.

We define the task of inline code comment generation as a conditional generation problem modeled by a language model M, such that:

$$M : \mathcal{X} \times \mathcal{L} \times \mathcal{I} \to \mathcal{Y} \quad \text{where} \quad M(x, l, i) = \hat{y}$$

Here, $\hat{y}$ is the generated code with inline comments, and the goal is for $\hat{y}$ to closely approximate the ground truth y in terms of both quality and quantity of generated comments.

In the zero-shot setting, $l = \emptyset$, and the model relies solely on the instruction i and the input code x. In few-shot settings, l includes multiple demonstration pairs to teach the intended transformation to the model M.

3.2 Inline Comments Evaluation Framework

An effective code documentation system must not only add meaningful and contextual comments to the code, but also discern the specific code blocks

that need explanation. The inline comments must be non-trivial, domain-aware and contribute to the understanding of the code block logic and functionality. Additionally, comment placement must be judicious: excessive commentary can clutter the code and impact readability, while sparse annotations risk omitting important code blocks that need explanation. Addressing this dual challenge requires an evaluation framework that is ideally language-agnostic and capable of assessing both the semantic relevance of comments and the appropriateness of their placement within the code snippets.

3.2.1 Comment scope

While generating $\hat{y}$, LLMs may inadvertently alter the original code, such as by introducing optimizations or unwrapping compact expressions, even when explicitly instructed not to do so. This behavior makes it unreliable to align comments between the original (y) and generated ($\hat{y}$) versions solely based on line numbers. Furthermore, as illustrated in Figure 1, discrepancies may arise in the granularity of comments where one version may contain multiple fine-grained annotations for a code block, while the other may offer a single, broader comment. To address such variations, we adopt a block-level comment matching strategy rather than a line-level alignment.

To perform a block-level comment matching procedure between y and $\hat{y}$, we first define the scope of an inline comment. In our framework, the scope extends from the comment line to either the next inline comment or the end of the current code block, determined usually by indentation levels in most programming languages. The second condition is particularly important, as not all code blocks are annotated; relying solely on the next comment could include unrelated, uncommented code, thereby introducing noise into the evaluation.

Using this definition, we parse both y and $\hat{y}$ code versions to identify corresponding comment-code pairs at the block level. For both the commented code versions y and $\hat{y}$, we record a mapping between each inline comment and its associated code scope, represented as a line range in the format:

`comment` $\rightarrow$ `[start_line_num, end_line_num]`

This mapping, recorded for both the versions separately, enables a fine-grained analysis of whether the model has over-commented or under-commented relative to the ground truth.

3.2.2 Comment alignment

Once the comment-to-scope mappings are established for both the reference code (y) and the generated version ($\hat{y}$), the next step is to align the inline comments across the two versions. This alignment is essential for enabling a fine-grained evaluation of documentation quality. To identify candidate pairs, we use the `start_line_num` and `end_line_num` of each comment's associated code block to detect scope overlaps between y and $\hat{y}$.

Given that the same code block may be annotated with varying levels of granularity, ranging from multiple fine-grained comments to a single high-level summary, we define four alignment cases that determine what gets included in the *comparison candidate set*:

- **Case 1 (Exact Match)**: If a comment from y (c_y) and a comment from $\hat{y}$ ($c_{\hat{y}}$) share an identical scope, the pair $(c_y, c_{\hat{y}})$ is directly added to the comparison set. These pairs contribute to the true positive count.

- **Case 2 (Partial Overlap)**: When the scopes of c_y and $c_{\hat{y}}$ partially overlap, typically due to differences in comment granularity, we aggregate all comments within the overlapping region from each version. For instance, a single c_y may align with a set of comments $\{c^1_{\hat{y}}, c^2_{\hat{y}}, \ldots\}$, or vice versa. These are concatenated in each version to form the composite comments:

  $$\left\{\text{Concat}\left\{c^1_y, c^2_y, \ldots\right\}, \text{Concat}\left\{c^1_{\hat{y}}, c^2_{\hat{y}}, \ldots\right\}\right\}$$

  This composite pair is then added to the comparison set. This strategy allows for flexibility in alignment, focusing on whether the code block is adequately explained rather than enforcing strict one-to-one comment matching. These pairs also contribute to the true positive count.

- **Case 3 (Missed by Model)**: If a comment c_y has no overlapping counterpart in $\hat{y}$, it is added to the comparison set as a false negative.

- **Case 4 (Hallucinated by Model)**: If a comment $c_{\hat{y}}$ has no overlapping counterpart in y, it is added to the comparison set as a false positive.

In summary, the comparison candidate set consists of all aligned comment pairs, either exact or

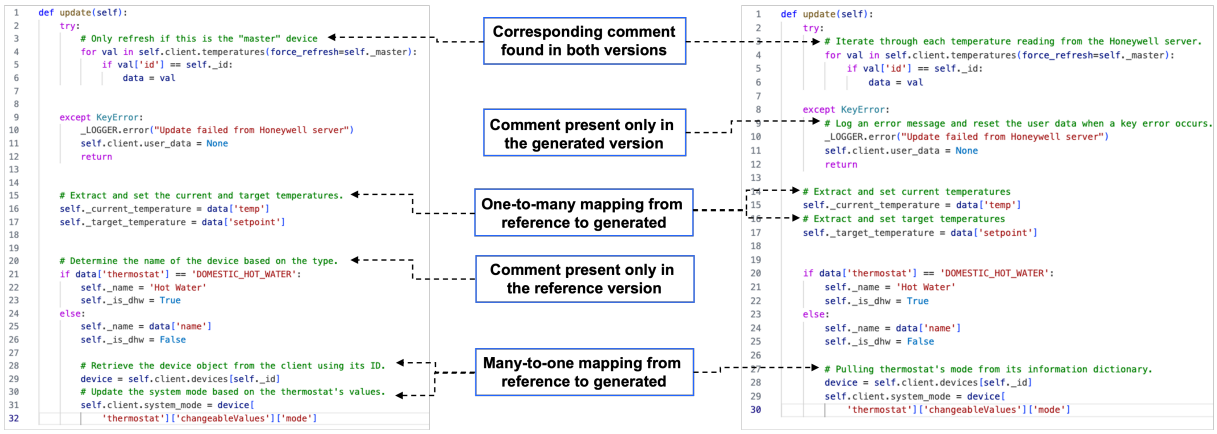

Figure 1: Illustrating comment alignment variations and representative mapping scenarios between reference and generated inline comments

aggregated, as well as unmatched comments from either version. This structured set forms the basis for evaluating the model's ability to generate contextually appropriate and well-placed inline documentation.

3.2.3 Quality metric

To assess the semantic relevance of the generated comments, we evaluate the aligned comparison candidates in terms of contextual similarity. This step is crucial for understanding how effectively an LLM interprets the underlying code logic and produces meaningful and *quality* documentation. Following the strategy used earlier for comment comparison (Geng et al., 2024; Szalontai et al., 2024; Sun et al., 2025a), we adopt an embedding-based approach to quantify this similarity. In particular, we employ a pretrained embedding model, SentenceTransformer's `all-MiniLM-L6-v2` (Reimers and Gurevych, 2021), to encode each comment in the aligned pairs and compute their similarity score. These scores are then aggregated at the sample level to yield an average similarity score per instance. We refer to this metric as $IC_{quality}$, which serves as an indicator of the interpretive and contextual fidelity of the generated comments with respect to the reference annotations.

3.2.4 Quantity metric

An often overlooked yet critical aspect of code readability is the documentation coverage. Striking the right balance in annotation density is essential: overly verbose comments can disrupt the cognitive flow of reading code, while insufficient documentation may leave key segments opaque to the reader. Existing approaches to evaluating comment generation systems predominantly focus on semantic relevance, frequently neglecting the quantification of sufficient documentation coverage.

To address this, our framework adds a *quantity* factor that measures block-level coverage equivalence between the reference y and generated $\hat{y}$. We use our comparison candidate set to compute the true positives, false positives and false negatives as outlined in Section 3.2.2. A key consideration in the evaluation of documentation system performance is the asymmetry in error impact: missing a comment on a developer-identified block (false negative) is more detrimental than over-commenting (false positive). To capture this notion, we propose to use f_β score, where β weighs the precision and recall contribution appropriately. For our study, we use $\beta = 2$ to value the recall more than the precision. We denote this metric as $IC_{quantity}$ capturing the adequacy of comment density in generated documentation.

3.2.5 Combined metric

To enable a holistic evaluation, we use a unified metric derived from the previously derived $IC_{quality}$ and $IC_{quantity}$ components. We compute a weighted average of these two values, allowing for flexible calibration based on task-specific priorities. The final evaluation score is given by:

$$\text{IC}_{\text{eval}} = w_1 \cdot \text{IC}_{\text{quality}} + w_2 \cdot \text{IC}_{\text{quantity}}$$

This formulation supports flexible evaluation across systems by adjusting the weights w_1 and w_2 to reflect different documentation goals. For our evaluation, we have given equal weightage to these components by setting $w_1 = 0.5$ and $w_2 = 0.5$

4 Experimental Setup

4.1 Dataset

To construct a high-quality benchmarking dataset for the task of inline comment generation, we begin with the train split of the Vault - Inline dataset, focusing exclusively on Python code samples. While the original dataset verifies the presence of inline comments, it does not account for their semantic quality or coverage across code blocks. To address this limitation, we apply a series of checks and quality filters aimed at curating a more representative and challenging dataset. Specifically, we retain only those functions that contain diverse programming constructs and are accompanied by meaningful, well-aligned inline comments. The resulting dataset, denoted hereafter as **'Vault-Inline++'**, serves as a robust benchmark for evaluating the performance of LLMs on the inline comment generation task.

The curation process for Vault-Inline++ dataset is explained in detail as follows:

- *Language checks*: The dataset contains code samples with multilingual inline comments. To ensure consistency and prevent distortion in evaluation metrics, we retain only those samples where comments are written entirely in English.

- *Content checks*: This step checks the content of comment in relation to the code that follows it, and eliminates those samples which may introduce noise. We exclude those samples which have decorative comments and samples where comment lines outnumber code lines.

- *Coverage of key programming constructs*: A critical requirement for evaluation is ensuring diverse and semantically rich code structures. To this end, we retain only those code samples that present a high density of inline comments across a variety of programming constructs. These include:

  – external function calls
  – conditional branches (e.g., if-else)
  – control flow statements like loops, break, continue, assert, etc.
  – exception handling blocks

  We leverage Abstract Syntax Tree (AST) parsing to identify the presence of these constructs and verify that each is accompanied by a corresponding developer-written comment.

- *Comment sufficiency*: As a final filtering step, we ensure that each code sample includes a sufficient volume of inline comments. Specifically, we retain only those samples where at least 10% of the code lines are accompanied by comments, and each comment meets a minimum word count threshold to ensure basic descriptive adequacy.

These filtering steps ensure that the final dataset includes code samples that have monolingual, consistent and detailed inline comments. Moreover, it also constitutes of programmatically rich and diverse samples with high volume of developer-annotated programming constructs. These samples form a robust test bed for evaluating the inline comment generation capabilities of language models. A few statistics on the final dataset are given in Table 1.

Measure	Value
Number of functions	2190
Average lines of code	70
Average length of comments	5

Table 1: Dataset composition used in our analysis.

4.2 Models

Language models finetuned on coding datasets, although smaller in size, have shown performance on par with larger, general-purpose foundational models across a range of code interpretation and generation tasks (Szalontai et al., 2024; Sun et al., 2025a). Models that have a deep understanding of programming language, structure, are better positioned to produce relevant and well-aligned comments. To draw meaningful conclusions about model suitability for code documentation systems, it is essential to conduct a fair comparison between smaller, code-finetuned models and larger foundational models.

In our study, we experiment with two foundational models - Anthropic's **Claude Sonnet 3.5** (Anthropic, 2024) and Meta's **Llama-3.1-70B** (AI, 2024) models - as representatives of larger general-purpose LLMs. For assessing the performance of code-finetuned models, we choose to evaluate Alibaba's **Qwen-Coder-2.5-1.5B** (Hui et al., 2024), Google's **CodeGemma-7B** (Team et al., 2024) and Meta's **CodeLlama-7B** (Rozière et al., 2023) models. We use the 'instruct' versions of these models,

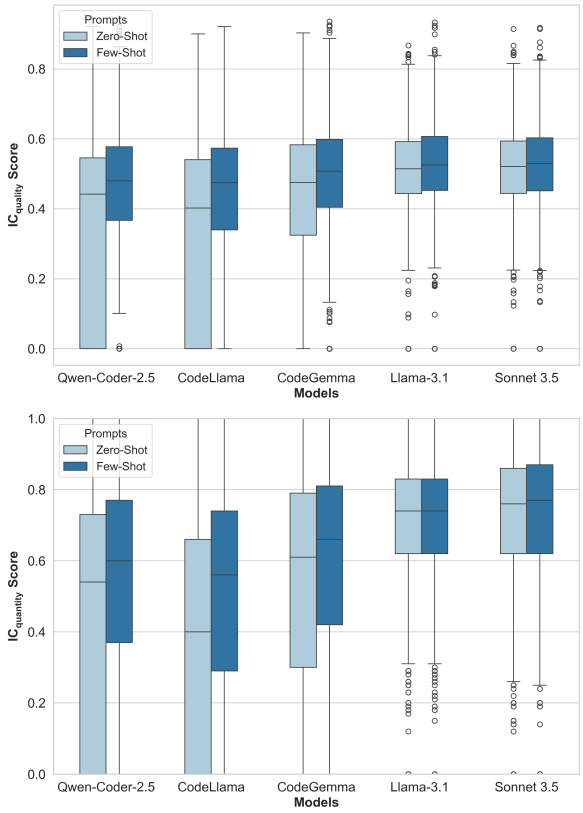

Figure 2: Comparative trends of $IC_{quantity}$ and $IC_{quality}$ scores under zero-shot and few-shot prompting settinfs

unless specified otherwise. The inference setting used while invoking each of these models is mentioned in Appendix A. Each of these models are trained on Python code samples and have shown strong performance on various coding benchmarks. Our choice of models span a wide spectrum in terms of training specialization and model sizes, enabling representative evaluation across different modeling paradigms and deployment scenarios.

4.3 Prompting Techniques

Most language models designed for code understanding and generation tasks are typically pretrained on curated code repositories and high-quality coding datasets (Kocetkov et al., 2022; Chaudhary, 2023). Since these corpora often include well-annotated code snippets, the language models possess a strong prior understanding of commented code. Hence, **zero-shot prompting** technique often suffices to instruct these models for generating meaningful comments.

However, the ability to determine which code blocks need to be commented requires logical reasoning, that can benefit from additional learning signals. Towards this, we experiment with **k-shot prompting** technique, where each shot is a pair of raw code snippet paired with its corresponding well-commented code version. To curate a rich bank of exemplars, we start with our Vault-Inline++ dataset and use *LLM-as-a-judge strategy* to help identify the ideal code samples that demonstrate a good balance of contextual comments with optimal quantity. The choice of model is driven by the fact that identifying such samples is a reasoning task as the judge needs to evaluate the relevance and impact of comments. Specifically, we employ Anthropic's Claude Sonnet 3.7-Thinking model (Anthropic, 2025) and instruct it to qualify each sample into *positive* or *negative* category. Among the *positive*-ly qualified samples, we choose top-n samples that have the highest density of the key programming constructs like function calls, conditional statements and exception handling blocks to ensure good diversity in our exemplar bank.

During inference with few-shot prompting, we use dynamic example selection strategy (Liu et al., 2022; Li et al., 2024; Bhattacharya and Gupta, 2024) to identify the most relevant examples based on code similarity. For each test instance, we compute similarity scores between its embedding and those of samples in the exemplar bank. These embeddings are obtained using the GraphCodeBERT (Guo et al.) model, which is pretrained to capture structural and semantic properties of source code. The top-k most similar examples are then selected as demonstration pairs to guide the model during generation. In our experiments, we fix $k = 3$ and maintain an exemplar bank of size $n = 50$.

5 Results

The experiments for generating commented code were conducted using the prompting strategies outlined in Section 4.3. The specific instructions and prompt templates provided to the models are detailed in Appendix C. This section presents the outcomes of these experiments and addresses the research questions defined earlier.

5.1 Main Findings

To ensure a fair evaluation, we first preprocess the raw outputs by correcting any code modifications introduced by the models. As proposed in our evaluation framework in Section 3.2, we compute three metrics: $IC_{quality}$, $IC_{quantity}$, and IC_{score}, which respectively assess semantic relevance, comment

density, and an aggregate performance measure. Table 2 presents a comparative overview of the scores across all evaluated models. Among the models evaluated, Claude Sonnet 3.5 consistently outperforms others across individual metrics. Notably, all models exhibit marked improvements under few-shot prompting conditions.

RQ1: Overall performance across models Figure 2 illustrates the distribution of scores obtained across the evaluated metrics. Larger foundational LLMs demonstrate consistently strong performance on the overall score, suggesting a robust capacity for producing high-quality inline code documentation. The notably high values for $IC_{quantity}$ across models indicate that LLMs are effective at identifying key code segments and inserting comments at appropriate locations. Furthermore, despite the variability in intent and style within the reference comments, the elevated $IC_{quality}$ scores suggest that the generated comments are semantically aligned with the code functionality and comparable to those written by developers.

RQ2: On Code-Specialized Models Code-specialized language models exhibit competitive performance on the combined metric relative to larger foundational models. However, their performance showcases greater fluctuations across different code samples. Among these, CodeGemma-7B stands out for maintaining a balanced trade-off between mean performance and variance across both metrics. Interestingly, Qwen-Coder-2.5-1.5B, despite being the smallest model in the cohort, delivers respectable average performance, making it a promising candidate for deployment in low-compute environments. Given that our selection of code models was guided by practical constraints suitable for industry-scale deployment, these results highlight the potential of such models to support in-house code documentation systems tailored to specific organizational styles, conventions, and requirements.

RQ3: Impact of Few-Shot Prompting The inclusion of few-shot exemplars in the prompt consistently elevates the overall performance metrics across models. While the improvement for larger foundational models remains relatively marginal, its impact on smaller, code-specialized models is both substantial and consistent. Specifically, few-shot prompting leads to a marked increase in mean performance and a notable reduction in variance, indicating that these models not only perform better on average but also exhibit greater stability across diverse code samples. This effect is particularly pronounced in models such as Qwen-Coder-2.5-1.5B and CodeLlama-7B, with the former outperforming the latter across all evaluation metrics despite its smaller size. These findings underscore the value of carefully curated exemplar pairs, especially for low-compute deployment scenarios. In such settings, investing in high-quality prompt design can yield significant gains in both the effectiveness and reliability of automated code documentation systems.

5.2 Instruction Adherence and Comment Coverage

One notable limitation observed in smaller code-specialized models is their inconsistent adherence to the provided instructions. For many test samples, these models generate only a high-level function docstring while copying the remainder of the input code verbatim, or they omit inline comments for critical code blocks altogether. This behavior results in poor alignment with the intended comment placement, as reflected by low $IC_{quantity}$ scores during our evaluation. In contrast, larger foundational models demonstrate better instruction adherence, even under zero-shot settings. To assess whether few-shot prompting mitigates this issue, we analyzed the number of samples that obtained low $IC_{quantity}$ scores in this setting. As shown in Figure 3, this number decreases substantially for the smaller models when few-shot exemplars are included in the prompt, but they still exhibit occasional failures despite that. Some of the examples with improved instruction adherence are also provided in Appendix B. For the larger models, there is little to no change in the quantity-based scoring.

5.3 Distributional Shifts in Semantic Quality

To assess the semantic quality of generated inline comments, we conducted a comparative analysis of samples positioned at the extremes of the $IC_{quality}$ spectrum - those rated as very poor versus those rated as good. We discretized the $IC_{quality}$ scores into three bins: poor, average, and good, using empirically derived thresholds based on the distribution across the test set. This allowed us to examine how model performance shifts under zero-shot and few-shot prompting conditions, particularly at the tails of the distribution. As illustrated in Figure 4, all models, including both foundational and code-

Table 2: Comparative evaluation of foundational and code-specialized language models on quantity ($IC_{quantity}$), quality ($IC_{quality}$), and composite (IC_{score}) metrics under zero-shot and few-shot prompting regimes.

	Model Size	IC_{quantity}		IC_{quality}		IC_{score}	
	# of Params (B)	Zero-Shot	Few-Shot	Zero-Shot	Few-Shot	Zero-Shot	Few-Shot
Qwen-Coder-2.5-1.5B	1.5	0.447 ± 0.325	0.53 ± 0.29	0.345 ± 0.241	0.411 ± 0.21	0.396 ± 0.265	0.471 ± 0.229
CodeLlama-7B	7	0.369 ± 0.319	0.498 ± 0.296	0.309 ± 0.253	0.402 ± 0.221	0.339 ± 0.268	0.45 ± 0.237
CodeGemma-7B	7	0.517 ± 0.32	$\mathbf{0.573 \pm 0.292}$	0.391 ± 0.233	$\mathbf{0.438 \pm 0.213}$	0.454 ± 0.256	$\mathbf{0.506 \pm 0.23}$
Llama-3.1-70B	70	0.704 ± 0.165	0.707 ± 0.169	0.489 ± 0.129	$\mathbf{0.501 \pm 0.132}$	0.597 ± 0.12	0.604 ± 0.123
Claude Sonnet 3.5	-	0.72 ± 0.174	$\mathbf{0.721 \pm 0.176}$	0.491 ± 0.131	0.498 ± 0.133	0.605 ± 0.123	$\mathbf{0.61 \pm 0.126}$

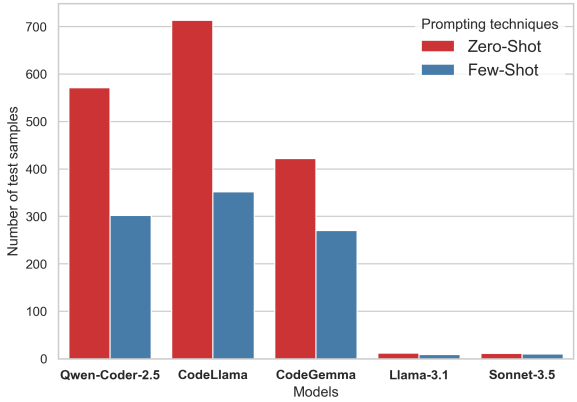

Figure 3: Comparison of test instances with $IC_{quantity} = 0$ across models under zero-shot and few-shot prompting settings

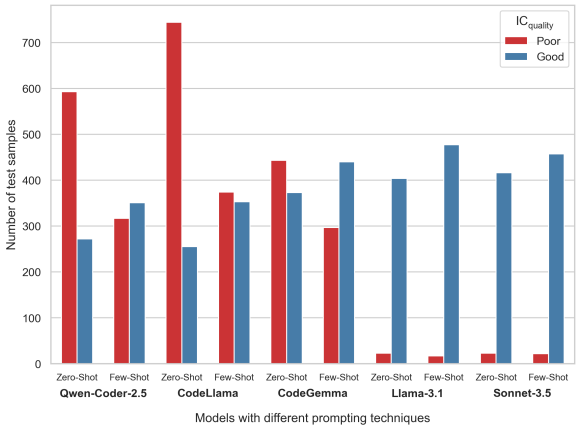

Figure 4: Comparison of low- and high-quality semantically relevant matches across models under zero-shot and few-shot prompting conditions

specialized groups, show consistent gains in the proportion of samples falling into the 'good' category, with improvements ranging from 9% (for Sonnet-3.5) to 38% (for CodeLlama). Notably, the incidence of 'poor' cases declines sharply for smaller models under few-shot settings. These findings suggest that the inclusion of well-crafted exemplars in the prompt substantially enhances the contextual relevance of generated comments, regardless of model size.

6 Conclusion

This work presents a comprehensive evaluation of large language models for inline comment generation, a task requiring both semantic precision and contextual coverage. Using a curated dataset of well-commented code, we propose a structured framework that enables holistic validation of generated comments under varied prompting conditions.

Our benchmarking reveals that larger foundational models consistently produce high-quality comments, while smaller, code-specialized models perform competitively with few-shot prompting. Exemplar-based prompts notably improve instruction adherence and output consistency, making smaller models strong candidates for low-compute environments where efficiency and adaptability are essential.

A key contribution of this work is an evaluation framework, enabling interpretable and fine-grained assessment of inline comments by jointly capturing semantic relevance and coverage. As high-quality annotations become increasingly available, such automated frameworks are vital for scalable benchmarking. Our findings highlight the importance of prompt design and model choice laying a foundation for future research in code-focused NLP.

Future work can extend this study by evaluating model performance across a wider range of programming languages to assess generalizability. It can also explore validation mechanisms for production systems that generate comments without ground-truth annotations, focusing on scalable methods to assess comment quality and coverage in real-world deployments.

References

2025. Ollama — local large model framework. https://ollama.org/. Open-source framework for running large language models locally, accessed: 2025-08-04.

Meta AI. 2024. Introducing llama 3.1: Our most capable models to date. Meta AI blog post. Accessed: 2025-07-25.

Anthropic. 2024. Claude 3.5 sonnet model card addendum. https://paperswithcode.com/paper/claude-3-5-sonnet-model-card-addendum. Accessed: 2025-07-25.

Anthropic. 2025. Claude 3.7 Sonnet System Card. PDF document on Anthropic website. Accessed: 2025-07-25.

Suborno Deb Bappon, Saikat Mondal, and Banani Roy. 2024. Autogenics: Automated generation of context-aware inline comments for code snippets on programming q&a sites using llm. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 24–35. IEEE.

Paheli Bhattacharya and Rishabh Gupta. 2024. Selective shot learning for code explanation. *arXiv e-prints*, pages arXiv–2412.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A comparative analysis of large language models for code documentation generation. In *Proceedings of the 1st ACM international conference on AI-powered software*, pages 65–73.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, and 1 others. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Rajarshi Haldar and Julia Hockenmaier. 2024. Analyzing the performance of large language models on code summarization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 995–1008, Torino, Italia. ELRA and ICCL.

Yuan Huang, Hanyang Guo, Xi Ding, Junhuai Shu, Xiangping Chen, Xiapu Luo, Zibin Zheng, and Xiaocong Zhou. 2023. A comparative study on method comment and inline comment. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–26.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2.5–coder technical report. *arXiv preprint arXiv:2409.12186*.

Dmitrii Kocetkov, Canwen Xu, Niklas Muennighoff, Baolin Peng, Georges Abdelnour, and 1 others. 2022. The stack: 3 tb of permissively licensed source code. In *Proceedings of the 7th Workshop on Representation Learning for NLP (RepL4NLP)*.

Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. Acecoder: An effective prompting technique specialized in code generation. *ACM Trans. Softw. Eng. Methodol.*, 33(8).

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online. Association for Computational Linguistics.

Dung Nguyen, Le Nam, Anh Dau, Anh Nguyen, Khanh Nghiem, Jin Guo, and Nghi Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 4763–4788, Singapore. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2021. Sentence-transformers: all-minilm-l6-v2. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2. Accessed: 2025-08-04.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Hwanjun Song, Hang Su, Igor Shalyminov, Jason Cai, and Saab Mansour. 2024. FineSurE: Fine-grained summarization evaluation using LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 906–922, Bangkok, Thailand. Association for Computational Linguistics.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025a. Source Code Summarization in the Era of Large Language Models . In *2025*

IEEE/ACM 47th International Conference on Software Engineering (ICSE), pages 1882–1894, Los Alamitos, CA, USA. IEEE Computer Society.

Weisong Sun, Yiran Zhang, Jie Zhu, Zhihui Wang, Chunrong Fang, Yonglong Zhang, Yebo Feng, Jiangping Huang, Xingya Wang, Zhi Jin, and 1 others. 2025b. Commenting higher-level code unit: Full code, reduced code, or hierarchical code summarization. *arXiv preprint arXiv:2503.10737*.

Balázs Szalontai, Gergő Szalay, Tamás Márton, Anna Sike, Balázs Pintér, and Tibor Gregorics. 2024. Large language models for code summarization. *arXiv preprint arXiv:2405.19032*.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, and 4 others. 2024. Codegemma: Open code models based on gemma. arXiv preprint arXiv:2406.11409.

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976.

A Model Inference settings

All language models used in this study, including both foundational LLMs and code-specialized variants, were inferred with a temperature setting of 10^{-8} ensuring near-deterministic outputs. The foundational models, namely the Claude series and Llama-70B, were accessed via Amazon Bedrock. The code-finetuned models were hosted locally through the Ollama framework (oll, 2025).

B Illustrative examples demonstrating improved adherence to instructions

This section presents pairs of outputs illustrating improved adherence to instructions when using k-shot prompting with a code-specialized smaller model.

```python
def get_tweets(self, handle):
    """
    Retrieves tweets for a given handle.
    Args:
        self: The instance of the class.
        handle: The Twitter handle to retrieve tweets for.
    Returns:
        A tuple containing a list of tweets and a string containing all tweet text.
    """
    tweets = []
    all_text = ''
    try:
        fetched_tweets = self.search_for_tweets_by_handle(handle, 100)['statuses']
        for tweet in fetched_tweets:
            parsed_tweet = {}
            if tweet['text'] != 'RT':
                parsed_tweet['text'] = tweet['text']
                parsed_tweet['sentiment'] = self.get_tweet_sentiment(tweet['text'])
                if  tweet['retweet_count'] > 0:
                    if parsed_tweet not in tweets:
                        tweets.append(parsed_tweet)
                else:
                    tweets.append(parsed_tweet)
                all_text += self.clean_tweet(tweet['text'])
        return tweets, all_text
    except tweepy.TweepError as e:
        print("Error : " + str(e))
```

(a)

```python
def get_tweets(self, handle):
    """
    Returns a list of tweets and a concatenated string of tweet text.
    INPUT:
        handle: The Twitter handle to retrieve tweets for.
    OUTPUT:
        tweets: A list of dictionaries containing tweet text and sentiment.
        all_text: A string containing the concatenated text of all tweets.
    """
    # Initialize a list to store tweets and a string to hold concatenated text
    tweets = []
    all_text = ''
    try:
        # Fetch tweets from Twitter API
        fetched_tweets = self.search_for_tweets_by_handle(handle, 100)['statuses']
        # Iterate through fetched tweets
        for tweet in fetched_tweets:
            # Parse tweet into a dictionary
            parsed_tweet = {}
            # Check if tweet is not a retweet
            if tweet['text'] != 'RT':
                # Add tweet text and sentiment
                parsed_tweet['text'] = tweet['text']
                parsed_tweet['sentiment'] = self.get_tweet_sentiment(tweet['text'])
                # Check if tweet is a retweet
                if  tweet['retweet_count'] > 0:
                    # Only add unique retweets
                    if parsed_tweet not in tweets:
                        tweets.append(parsed_tweet)
                else:
                    # Add all other tweets
                    tweets.append(parsed_tweet)
                # Concatenate tweet text
                all_text += self.clean_tweet(tweet['text'])
        # Return list of tweets and concatenated text
        return tweets, all_text
    # Handle errors
    except tweepy.TweepError as e:
        print("Error : " + str(e))
```

(b)

```python
def download_template_files(self):
    """
    Downloads template files for the given frontend library or framework.
    Args:
        self: The object instance.
    """
    template_files = constants.PROD_TEMPLATE_FILES_DICT.get(
        self.frontend_library_or_framework
    )
    substitute_parameters = {"app_name": self.app_name}
    for directory_type, filename, download_url in template_files:
        target_filepath = self.get_target_path_of_template_file(
            filename, directory_type
        )
        download_file = requests.get(download_url, stream=True)
        with open(target_filepath, "wb") as target_file:
            try:
                target_file.write(download_file.content)
            except OSError as error:
                raise CommandError(error)
        if not self.check_if_file_is_text_document(filename):
            continue
        with open(target_filepath, "r") as target_file:
            source_file = Template(target_file.read())
            modified_file_contents = source_file.substitute(substitute_parameters)
        with open(target_filepath, "w") as target_file:
            try:
                target_file.write(modified_file_contents)
            except OSError as error:
                raise CommandError(error)
```

(c)

```python
def download_template_files(self):
    """
    Downloads template files for the given frontend library or framework.
    """
    # Get the template files for the given frontend library or framework
    template_files = constants.PROD_TEMPLATE_FILES_DICT.get(
        self.frontend_library_or_framework
    )
    # Define the parameters to substitute in the template files
    substitute_parameters = {"app_name": self.app_name}
    # Iterate over the template files
    for directory_type, filename, download_url in template_files:
        # Get the target filepath for the template file
        target_filepath = self.get_target_path_of_template_file(
            filename, directory_type
        )
        # Download the template file
        download_file = requests.get(download_url, stream=True)
        # Save the template file
        with open(target_filepath, "wb") as target_file:
            try:
                target_file.write(download_file.content)
            except OSError as error:
                raise CommandError(error)
        # Check if the template file is a text document
        if not self.check_if_file_is_text_document(filename):
            continue
        # Open the template file for reading
        with open(target_filepath, "r") as target_file:
            # Create a template object
            source_file = Template(target_file.read())
            # Substitute the parameters in the template file
            modified_file_contents = source_file.substitute(substitute_parameters)
        # Open the template file for writing
        with open(target_filepath, "w") as target_file:
            try:
                target_file.write(modified_file_contents)
            except OSError as error:
                raise CommandError(error)
```

(d)

Figure 5: Demonstration pairs (a), (b) and (c), (d) illustrating improved instruction adherence with k-shot prompting. Panels (a) and (c) show outputs under zero-shot prompting, while (b) and (d) present the corresponding k-shot generations with CodeGemma model.

C Prompt templates

For reproducibility, we provide below the prompt templates that were used to query LLMs for the respective tasks.

Prompt to employ LLM-as-a-judge for qualifying an exemplar

You are an experienced developer and code reviewer. You are given a code snippet or a function with inline comments added by a developer. Your task is to carefully analyze the inline comments written in this code, and based on that, categorize the code into one of the two categories - positive or negative. Follow up your answer with a proper justification of why the code was categorized into the final category. Make sure that the given rules are strictly followed. Be stricter while making your decision.
Follow the given rules STRICTLY while categorizing the code:
<rules>
Positive category
- Most of the important code blocks are properly commented. The code has balanced number of inline comments.
- Inline comments are explanatory and contextual, helping the reader to understand the code functionality.
- Most of the comments are high quality and contextual.
Negative category
- Either too many comments are present, or a lot of important code blocks have no comments written for them.
- Inline comments are too generic and naive, and do not add any value to code interpretation.
</rules>

Follow the given output format while responding. Do not add any additional lines or explanations:
<output_format>
Reason: reason for categorizing the code into the final category
Category: Positive or Negative
</output_format>

Now analyze and categorize the following code:
{input_code}

Zero-shot prompt for generating inline comments

You are an experienced Python developer who is responsible for maintaing the documentation and comments in the codebase. Given a Python code snippet or a function as input which consists of barely any comments, your goal is to add inline comments to the code and convert it into a well-commented coversion. All your comments must be meaningful and context-aware such that any junior developer can read them and understand the code functionality. You are only allowed to add comments to the input code, without modifying the existing code lines.

Follow the given guidelines while adding your inline comments:
<guidelines>
- Identify important blocks or set of code lines and add comments for them. Do not add comments for simpler lines of code, and do not leave any major block uncommented. Strike a balance in your response.
- Your comments must be highly contextual and meaningful to the domain for which the code is written.
- Do not add trivial or naive comments as they are not really helpful in code understanding.
- Add appropriate comments for every function call that is present in the code.
- Add appropriate comments for every if-else, loop, assert, break or similar code flow altering statements.
- Add appropriate comments for exception or error handling blocks.
- Add comments only on top of a code line. Do not add comments in front of the line.
- Return the commented version of the same code enclosed in triple backticks in your response. Do not add any additional lines or explanations.
</guidelines>

Now write a commented version for the following code:
{input_code}

k-shot prompt for generating inline comments

You are an experienced Python developer who is responsible for maintaing the documentation and comments in the codebase. Given a Python code snippet or a function as input which consists of barely any comments, your goal is to add inline comments to the code and convert it into a well-commented coversion. All your comments must be meaningful and context-aware such that any junior developer can read them and understand the code functionality. You are only allowed to add comments to the input code, without modifying the existing code lines. Use the provided examples as reference to understand how a commented code version looks like.

Follow the given guidelines while adding your inline comments:
<guidelines>
- Identify important blocks or set of code lines and add comments for them. Do not add comments for simpler lines of code, and do not leave any major block uncommented. Strike a balance in your response.
- Your comments must be highly contextual and meaningful to the domain for which the code is written.
- Do not add trivial or naive comments as they are not really helpful in code understanding.
- Add appropriate comments for every function call that is present in the code.
- Add appropriate comments for every if-else, loop, assert, break or similar code flow altering statements.
- Add appropriate comments for exception or error handling blocks.
- Add comments only on top of a code line. Do not add comments in front of the line.
- Return the commented version of the same code enclosed in triple backticks in your response. Do not add any additional lines or explanations.
- Use the given list of examples as reference to understand how inline comments are added by developers to form a commented version.
</guidelines>

Use the given example pairs of inputs and outputs for your reference:
<examples>
{list_of_fewshots}
</examples>

Now write a well-commented version for the following code:
{input_code}