ReFEree: Reference-Free and Fine-Grained Method for Evaluating Factual Consistency in Real-World Code Summarization

Suyoung Bae, CheolWon Na, Jaehoon Lee, Yumin Lee, YunSeok Choi*, Jee-Hyong Lee*
College of Computing and Informatics
Sungkyunkwan University, South Korea
{sybae01, ncw0034, hoon1223, totoandy, ys.choi, john}@skku.edu

Abstract

As Large Language Models (LLMs) have become capable of generating long and descriptive code summaries, accurate and reliable evaluation of factual consistency has become a critical challenge. However, previous evaluation methods are primarily designed for short summaries of isolated code snippets. Consequently, they struggle to provide fine-grained evaluation of multi-sentence functionalities and fail to accurately assess dependency context commonly found in real-world code summaries. To address this, we propose **ReFEree**, a reference-free and fine-grained method for evaluating factual consistency in real-world code summaries. We define factual inconsistency criteria specific to code summaries and evaluate them at the segment level using these criteria along with dependency information. These segment-level results are then aggregated into a fine-grained score. We construct a code summarization benchmark with human-annotated factual consistency labels. The evaluation results demonstrate that ReFEree achieves the highest correlation with human judgment among 13 baselines, improving 15-18% over the previous state-of-the-art. Our code and data are available at https://github.com/bsy99615/ReFEree.git.

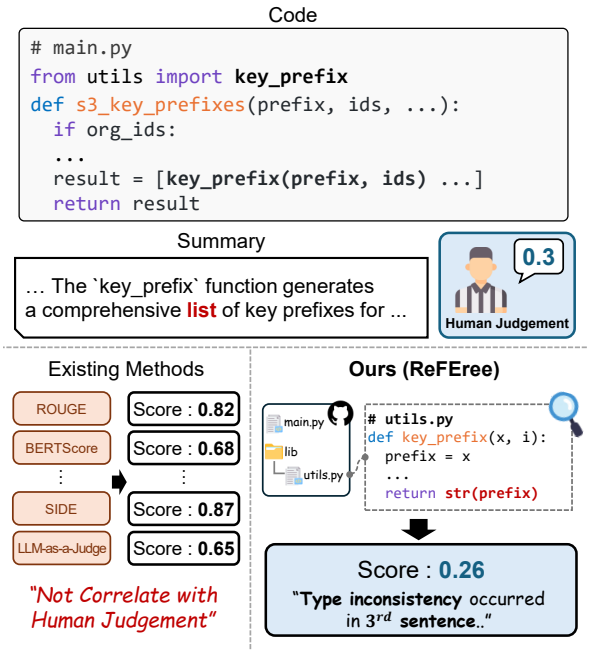


Figure 1: When evaluating factual consistency in real-world code summaries, existing methods (**bottom left**) fail to capture that the explanation of the 'key_prefix' in the summary exhibits a type error. In contrast, our method (**bottom right**) uses segment-level evaluation with related 'key_prefix' information, better aligning with human judgment.

1 Introduction

Recent advances in Large Language Models (LLMs), such as GPT-4, have made it feasible to automatically generate long and descriptive code summaries (Achiam et al., 2023; Sun et al., 2024). LLM-powered assistants such as OpenAI's Codex (Chen et al., 2021), GitHub Copilot (GitHub, 2021), and Anthropic's Claude-Code (Anthropic, 2025) are increasingly integrated into real-world development workflows to assist engineers in understanding and reviewing code.

However, when the generated summary does not accurately reflect the code's actual implementation, it can cause developers to misunderstand the code, leading to delayed debugging and increased maintenance costs (Wen et al., 2019; Jiang and Hassan, 2006; Ibrahim et al., 2012). Therefore, it is important to accurately and reliably evaluate the factual consistency of LLM-generated code summaries.

The existing reference-based metrics, such as ROUGE (Lin, 2004), BLEU (Papineni et al., 2002), and METEOR (Banerjee and Lavie, 2005), require human-written reference summaries and measure the lexical overlaps. However, code summarization is a one-to-many task meaning that semantically accurate summaries may use various wording from the reference (Naik et al., 2024; Wu et al., 2024).

*Corresponding authors

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 14352–14377

Consequently, reference-based evaluation methods are limited in detecting factual inaccuracies in such code summaries.

Recent approaches use LLMs as judges, taking the original code and its summary as inputs to assign a consistency score based solely on the LLM's internal knowledge (Zheng et al., 2023; Liu et al., 2023; Wu et al., 2024). These methods have the advantage of efficiently scoring the code summaries without references or a training process. However, they treat the code summary as a whole and consider factual consistency under a single criterion, producing only a binary or coarse-grained 5-point scale score. This oversimplified approach has two critical limitations. First, it cannot provide fine-grained assessments of inconsistencies in long summaries. Each sentence may contain different levels or types of factual errors, but identifying these nuances is lost in a single criterion score. Second, they fail to identify which specific sentences are inconsistent or explain the underlying reasons for the inconsistencies, making it difficult to refine and improve the LLM-generated code summary.

Additionally, in real-world code, functions or classes referenced within the input code are often defined externally, and code summaries frequently describe such external elements (Li et al., 2024; Ding et al., 2024; Liu et al., 2024b). However, existing methods evaluate summaries based solely on the input code without considering this external context, making them unable to accurately assess whether descriptions of externally defined elements are factually consistent. For example, as shown in Figure 1, when the summary describes the functionality of '`key_prefix`', evaluating its accuracy requires not only the input code but also the external context where it is defined.

Therefore, we propose a novel evaluation method, ***ReFEree***, a reference-free and fine-grained method for evaluating the factual consistency of code summarization in a real-world environment. This method is built upon four representative criteria that should be considered when evaluating real-world code summaries. Based on these criteria, a segment-level evaluation approach is designed to localize factual inconsistencies and provide actionable feedback. Furthermore, a code-related information searching mechanism is employed to accurately evaluate summaries containing external information unavailable in the input code alone. Our framework combines fine-grained evaluation with explicit code-related evidence, ensuring explainability of how the final consistency score is derived while enabling objective and consistent evaluation by minimizing reliance on internal knowledge.

To verify that *ReFEree* is reliable and accurate, we construct an evaluation benchmark that includes human labels of factual consistency for LLM-generated code summaries at both the summary-level and the segment-level. We collect project codes from 125 real-world repositories written in Python and Java. Using this benchmark, we compare *ReFEree* with 13 existing evaluation methods. The results show that *ReFEree* achieves the highest correlation with human judgment for both languages. Furthermore, various additional experiments demonstrate that our evaluation method can consistently maintain evaluation performance across various environments and serve as a stable and generalizable evaluation.

2 Related Works

Factual Consistency in Code Summarization. Factual consistency is essential for evaluating the reliability of LLM-generated code summaries. While there has been considerable research on factual consistency in code generation (Liu et al., 2024a; Tian et al., 2025; Zhang et al., 2025), those in code summarization remain relatively underexplored. Kang et al. (2024) and Wen et al. (2019) define inconsistencies in code comments, but only focus on specific aspects such as design constraints and parameter types. Maharaj et al. (2025) propose a cause-oriented taxonomy of error types, but it is not directly applicable to detection. Furthermore, existing studies focus on isolated functions, overlooking inconsistencies involving external dependencies.

Evaluation of Code Summarization. Most prior studies have evaluated code summarization using metrics adapted from natural language summarization (Ahmad et al., 2020; Choi et al., 2021, 2023), broadly into two types. First, reference-based methods evaluate textual similarity between human references and generated summaries using n-gram overlap (BLEU (Papineni et al., 2002), ROUGE (Lin, 2004), METEOR (Banerjee and Lavie, 2005)) or embedding similarity (BERTScore (Zhang et al., 2020), Sentence-BERT (Reimers and Gurevych, 2019)). However, low-quality or outdated references can misrepresent the actual quality of the generated summary.

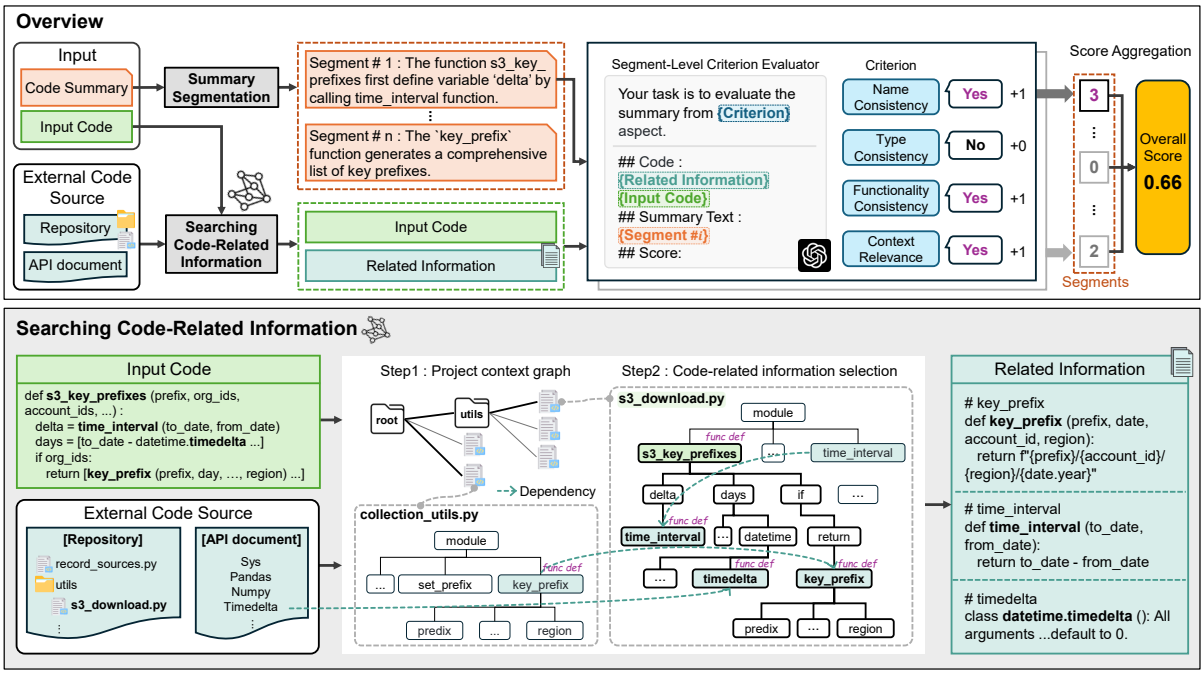


Figure 2: Overview of *ReFEree* framework. Our evaluation process first performs code-related information searching to consider dependency relations in the input code. The summary is then segmented at the sentence level, and an LLM evaluates each segment according to the four factual inconsistency criteria to obtain segment-level scores. These scores are subsequently aggregated to compute the final consistency score.

Second, reference-free methods directly compute similarity between the summary and the code without relying on references. SIDE (Mastropaolo et al., 2024) evaluates the semantic fitness by using contrastive learning to distinguish suitable and unsuitable summaries. However being trained on short summaries, it performs poorly for long, descriptive summaries covering multiple functionalities. Inspired by recent NLP advances, several approaches use LLMs as evaluators to simulate human judgment by measuring summary's factuality (Wang et al., 2023a; Liu et al., 2023; Min et al., 2023; Song et al., 2024). However, few studies have tailored these methods to the code domain. Wu et al. (2024) propose a judge prompt that assess a code summary as a whole using a single "factual consistency" criterion with 5-point scale score, limiting both explainability and granularity. Maharaj et al. (2025) attempt fine-grained evaluation at the entity level, but focus solely on binary inconsistency detection without explaining why an entity is incorrect, and rely on an LLM's internal knowledge without explicitly modeling external dependencies.

3 Methodology

In this section, we describe our evaluation method. We first explain the process of defining four factual inconsistency criteria (Section 3.1). As shown in Figure 2, *ReFEree* proceeds in two stages by first searching input code-related information (Section 3.2) and then calculating the overall score via fine-grained evaluation (Section 3.3).

3.1 Factual Inconsistency Criteria

To systematically evaluate the factual consistency of real-world code summaries, well-defined and generalizable criteria are required. Since LLM-generated summaries exhibit distinct error patterns, often stemming from hallucination or misinterpretation of code semantics, we empirically analyze the patterns of factual inconsistencies specific to LLM-generated summaries and identify four representative patterns to serve as evaluation criteria. We randomly sample 100 real-world Python and Java functions containing entities with dependency chains from popular repositories and generate descriptive summaries using three LLMs, Qwen-Coder, CodeLlama, and GPT-4 to ensure that the identified patterns are generalizable and not artifacts of a specific model's behavior. Three human annotators then review all 300 summaries and manually annotate any factual inconsistencies, with their comments (e.g., type errors, functionality inaccuracies).

Our analysis reveals four main patterns of factual

Criteria (Distribution)	Definition
[C1] Name Inconsistency (14%)	The name of a function, class, or variable in the summary does not match the actual identifier in the input code or mistakenly refers to a different entity with the same name in the project.
[C2] Type Inconsistency (15%)	The described function's return type or variable type in the summary is inconsistent with the actual type in the input code or in directly dependent functions.
[C3] Functionality Inconsistency (35%)	The functionality or purpose described in the summary does not accurately reflect what the input code or its dependent functions actually implement. This often arises when dependency relationships are ignored or misinterpreted, leading to an incorrect description of behavior.
[C4] Context Irrelevant (33%)	The summary contains content that is unnecessary or irrelevant to the input code or relevant information, such as descriptions of unrelated entities or overly generalized dependency context that does not contribute to understanding the function.

Table 1: Four factual inconsistency criteria with the proportion of each for evaluating real-world code summarization.

inconsistencies in LLM-generated code summaries. The most frequent is functionality inconsistency (35%), where explanations diverge from actual implementations due to misinterpreted dependency relations. The second most common is irrelevant context (33%), where the generated summary introduces functionality, purpose, or usage context that cannot be verified from the code itself and thus appears to be hallucinated. Unlike functionality inconsistency, which reflects a misreading of existing code behavior, irrelevant context represents a more deceptive form of factual error: ungrounded claims are presented as facts, directly distorting the developer's perception of what the code actually does.

The remaining patterns are name (14%) and type inconsistency (15%), involving incorrect descriptions of code element names or return types. The remaining 3% consist of minor issues, such as content duplication, that do not critically hinder code understanding.

Based on this analysis, we define four evaluation criteria specifically tailored for assessing the factual consistency of LLM-generated real-world code summaries. Table 1 presents these criteria along with their definitions. More detailed explanations and examples can be found in Appendix A.1.

3.2 Searching Code-Related Information

Real-world code summaries generated by LLMs often describe external elements such as function calls, class hierarchies, and API usage. Therefore, we adopt a two-step approach based on static program analysis approach to provide objective evidence for evaluating such external descriptions. Further details are provided in Appendix A.2.

Step 1: Project context graph construction. We first construct a project context graph. Given a project repository that contains the input code targeted for evaluation, we traverse the abstract syntax tree (AST) (Zhang et al., 2019) of each file in the repository to collect the code entities. Based on the parsed information, we construct a heterogeneous directed acyclic graph, where all types of code entities within the project are represented as nodes, and their relationships are linked as directed edges. Each node stores metadata such as the entity's AST type and code context, and each edge represents dependency relationships between two nodes, such as assignment, class inheritance, or function calls.

Step 2: Code-related information selection. Including all project context can produce noise that is not essential for understanding the core functionality (Lomshakov et al., 2024). To mitigate the interference of trivial or unnecessary relations in summary evaluation, we adopt *crucial entity selection* and a *1-hop dependency searching* strategy to selectively search only the dependency information that is essential and directly relevant to code summary evaluation.

We begin by performing a depth-first search (DFS) on the project context graph, starting from each entity node in the input code to traverse all related nodes connected via dependency edges. Among all related nodes, we first selectively search entities corresponding to 'functions', 'classes', and 'variables' that play a central role in real-world code understanding, as empirically determined through experiments at previous research (Cheng et al., 2024; Luo et al., 2024). Second, instead of exploring all dependency relations in the input code, we stop further searches once the dependency context of each core entity has been searched. This approach is based on prior research showing that multi-hop searches in hierarchical dependencies can increase noise for summary understanding as the number of hops increases (Zhang et al.,

2025). If an entity has external dependencies, we retrieve information from predefined API documentation (Ablation analysis comparing different n-hop search configurations is provided in Appendix C.1).

3.3 Calculating Factual Consistency Score

For fine-grained evaluation, we first apply the NLTK sentence tokenizer [1] to split a code summary ($\mathcal{D}$) into sentence-level segments, resulting in a set of segments $\mathcal{D} = \{S_1, S_2, ..., S_i, ..., S_n\}$.

We then evaluate each segment against the four criteria defined in Section 3.1. For evaluation, we construct a criterion evaluation prompt as briefly illustrated in Figure 2 (complete prompt in Appendix A.3), and use an LLM to determine whether each segment contains factual inconsistencies with respect to each criterion, considering both the input code and related information obtained in Section 3.2. The evaluator $f(S, C)$ outputs 0 if any factual inconsistency is detected in segment S with respect to criterion C, and 1 otherwise.

The total segment-level score is calculated by summing the outputs across all four criteria. Then, the segment-level scores are aggregated to calculate the final overall score. Given a summary segmented as $\mathcal{D}$ and a set of criteria $Criteria = \{C_1, C_2, C_3, C_4\}$, the overall factual consistency score is computed as follows:

$$SCORE = \frac{1}{|\mathcal{D}| \times |Criteria|} \sum_{S \in \mathcal{D}} \sum_{C \in Criteria} f(S, C)$$

The score ranges from 0 to 1, with higher values indicating greater factual consistency. This decompose-and-aggregate approach enables fine-grained assessment while ensuring interpretability of the overall scoring process.

4 Experiment Setups

4.1 Evaluation Benchmark

To verify the effectiveness of our proposed evaluation method, it is essential to demonstrate that its evaluation results using our method are highly correlated with human judgment. However, no existing benchmark provides human-annotated factual consistency labels for LLM-generated code summaries. Moreover, naturally generated LLM summaries lack sufficient hallucinated examples to rigorously test inconsistency detection capabilities. Therefore, we construct a reliable set of human labels for factual consistency at both the summary and segment levels. In this section, we briefly describe the labeling process in three steps; more detailed procedures, further analysis, statistics, and examples are provided in Appendix B.1.

Step 1: Source code collection. We build our benchmark by extracting project codes from real-world open-source repositories: 1,825 Python functions from DevEval (Li et al., 2024) (115 repositories) and 230 Java functions from ClassEval (Du et al., 2023) (10 repositories). Approximately 86% of the functions are non-standalone, containing context-aware dependencies. Both languages are prevalent in open-source environments and feature diverse dependency relationships, ensuring a wide range of code complexity.

Step 2: Summary generation. We use ChatGPT to generate long and descriptive code summaries, intentionally including at least one factually inconsistent sentence in each summary. We design a structured instruction prompt consisting of three components: an instruction description, a set of inconsistency patterns with demonstrations [2], and the target code along with related information, intentionally guiding the LLM to generate summaries that include inconsistency contents arising when such related information is not properly considered.

Step 3: Factual consistency labeling. We then annotate factual consistency ground truth labels at both summary and segment levels. At the summary level, each summary is labeled on a 5-point scale (1: highly inconsistent to 5: highly consistent). At the segment level, each sentence is labeled as `CORRESPOND` (1) or `NOT CORRESPOND` (0) to each of the four criteria (C1–C4) defined in Section 3.1.

To obtain reliable and accurate labels, we adopt a Human–AI collaborative labeling approach, which has proven effective in producing high-quality annotations (Li et al., 2023; Zhang et al., 2023; Kim et al., 2024). First, three LLMs with different roles determine the candidate label through majority voting. If all three LLMs may assign different scores during summary-level labeling (since there are five possible labels), the score closest to the mean is selected as the candidate label. Then, three human annotators subsequently post-edit the candidate label. The final label is determined according to the

[1] https://www.nltk.org/api/nltk.tokenize.html

[2] These patterns are derived from empirically observed error patterns identified in real-world summary generation scenarios, as analyzed in Section 3.1. Therefore, our generated summaries are designed to reflect the real-world distribution of mistakes that naturally occur.

Methods	Python				Java			
	r_p	r_s	τ	Average	r_p	r_s	τ	Average
reference-based methods								
ROUGE-1 (Lin, 2004)	0.046	0.044	0.034	0.041	0.201	0.186	0.140	0.176
ROUGE-2 (Lin, 2004)	0.030	0.031	0.024	0.028	0.194	0.176	0.137	0.169
ROUGE-L (Lin, 2004)	0.045	0.037	0.029	0.037	0.202	0.178	0.137	0.172
BLEU (Papineni et al., 2002)	0.016	0.018	0.015	0.017	0.081	0.065	0.058	0.068
METEOR (Banerjee and Lavie, 2005)	0.033	0.024	0.019	0.025	0.135	0.139	0.105	0.126
BERTScore (Zhang et al., 2020)	0.015	0.000	0.000	0.005	0.189	0.149	0.113	0.150
SBCS (Reimers and Gurevych, 2019)	0.068	0.036	0.028	0.044	0.363	0.227	0.178	0.256
SBED (Reimers and Gurevych, 2019)	-0.058	-0.036	-0.028	-0.041	-0.348	-0.227	-0.178	-0.251
reference-free methods								
SIDE (Mastropaolo et al., 2024)	-0.064	-0.056	-0.043	-0.054	0.032	0.024	0.019	0.025
LLM-judge (Zheng et al., 2023)	0.419	0.410	0.360	0.396	0.385	0.363	0.318	0.355
G-Eval (Liu et al., 2023)	0.427	0.413	0.360	0.400	0.457	0.407	0.355	0.406
Factscore (Min et al., 2023)	0.410	0.426	0.338	0.391	0.391	0.365	0.310	0.355
CODERPE (Wu et al., 2024)	0.418	0.405	0.353	0.392	0.457	0.398	0.347	0.401
ReFEree (w/o info)	0.432	0.432	0.349	0.404	0.469	0.458	0.387	0.438
***ReFEree* (w/ info)**	**0.497**	**0.489**	**0.390**	**0.459**	**0.515**	**0.502**	**0.423**	**0.480**

Table 2: We compare our method (*ReFEree*) with 8 reference-based and 5 reference-free methods using Pearson (r_p), Spearman (r_s), and Kendall's Tau (τ) correlation coefficients at the summary level. Our method achieves p-values all below 0.005, indicating statistical significance. The best result is shown in **bold**.

following three rules: (1) If at least two annotators agree with the candidate label, the label is retained. (2) If at least two annotators revise to the same label, the label is edited accordingly. (3) For summary-level labeling: If all annotators assign different labels, the final label is determined through further discussion among the three annotators.

We compute Krippendorff's alpha reliability (Krippendorff, 2018) to assess annotator agreement. We achieve an agreement of 0.74 at the summary level and an average of 0.84 at the segment level, both indicating a high level of agreement. All three human annotators hold at least a Master's degree in Computer Science and have an average of over four years of experience in Python and Java programming, ensuring the expertise and credibility of the annotations.

4.2 Baselines

We select 13 commonly used evaluation methods, 8 reference-based and 5 reference-free methods, from code summarization tasks to compare with our proposed method. Detailed explanations are provided in Appendix B.2.

4.3 Implementation Details

Our method supports various LLMs, including both closed and open-source models, as segment-level criterion evaluators. In our main experiments, we use OpenAI's GPT-4.1-mini to ensure a fair comparison with existing baselines under the same setting. All evaluations are conducted with consistent hyperparameters: temperature = 0.1, top-p = 0.9, top-k = 50, and max new tokens = 4. All experiments are conducted three times with different seeds, and the average results are reported. Evaluating the code summary using *ReFEree* incurs a cost of only $0.004 per sample. A detailed comparison of time and cost with existing methods is provided in Appendix C.2.

5 Results and Analyses

5.1 Comparisons with Baselines

To verify whether our method exhibits a high correlation with human judgment, we compute the Pearson (r_p), Spearman (r_s), and Kendall's tau (τ) correlation coefficient between summary-level factual consistency scores by humans and those produced by each method. Table 2 presents the main results of this comparison.

ReFEree outperforms all baselines. *ReFEree* achieves the highest correlation with human judgment in both Python and Java environments. Compared to G-Eval, the strongest baseline, *ReFEree* achieves approximately 15% (Python) and 18% (Java) higher correlation. FactScore decomposes

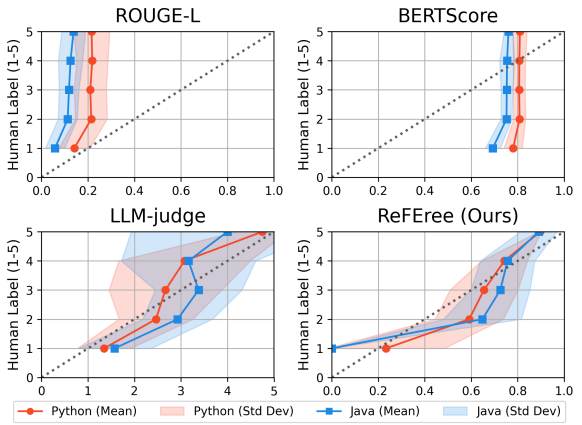


Figure 3: Correlation between human judgment and the scores from each method. Each plot shows the relationship between scores obtained from each method **(x-axis)** and human labels **(y-axis)**, with red for Python and blue for Java. The gray dashed line represents perfect correlation. Since human labels range from 1 to 5, there are no values at y = 0.

Criteria	C1	C2	C3	C4	Average
Python	0.922	0.961	0.931	0.924	0.934
Java	0.913	0.991	0.940	0.876	0.930

Table 3: Accuracy of segment-level factual inconsistency evaluation. We report the accuracy of *ReFEree* in identifying four criteria (C1–C4) defined in Section 3.1.

Code

```
def convert_to_idn(url):
    parts = list(urllib.parse.urlsplit(url))
    try:
        parts[1].encode('ascii')
    except UnicodeEncodeError:
        // Some code . . .
        if port:
            parts[1] += ':' + port
            return urllib.parse.urlunsplit(parts)
        else:
            return url
```

Human Label : 0.60 (Normalized)

ROUGE-L : 0.18 BERTScore : 0.81
SIDE : 0.85 G-Eval : 0.40 (Normalized)

***ReFEree* score : 0.55**

Summary : The 'convert_to_idn' function is designed to convert the hostname portion of a URL into its Internationalized Domain Name (IDN) ASCII-compatible encoding using the Punycode standard. It first splits the URL into its components using the 'urlparse' function , then attempts to encode the hostname in ASCII **(C1, C3)**. [Some explanations . . .] The function returns a list of URL components with the converted hostname **(C2, C3)**. This utility is particularly useful in web crawlers and multilingual domain management systems where URLs need to be normalized for consistent processing and storage **(C4)**.

Table 4: Qualitative comparison of *ReFEree* and existing methods. Factually inconsistent segments are highlighted in purple, with C1–C4 indicating the violated criteria (defined in Table 1). Human label and G-Eval scores are normalized to 0~1 for comparison. More comparisons are described in Appendix C.3.

summaries into atomic facts but evaluates each with only a single consistency criterion. *ReFEree*, by contrast, outperforms FactScore by 17.4% (Python) and 35.2% (Java) through multi-criteria segment-level evaluation specifically designed for real-world code summaries.

We further conduct an ablation study to assess the impact of code-related information. When related information is excluded (*ReFEree* (w/o info)), correlation with human judgment decreases by approximately 0.05 in both Python and Java. This demonstrates that our information searching module plays a vital role in achieving accurate factual consistency evaluation.

To further analyze how each method evaluates factual consistency in comparison to human judgment, we compare the score distributions of ROUGE-L, BERTScore, LLM-judge, and *ReFEree*. Figure 3 visualizes the relationship between evaluation scores and human labels by plotting the mean and standard deviation of scores corresponding to each human label. The observed patterns reflect inherent characteristics of each method. ROUGE-L consistently assigns low scores even to factually consistent summaries due to its reliance on lexical overlap, while BERTScore tends to assign high scores even to summaries with inaccurate descriptions. LLM-judge show moderate correlation with human judgment but exhibit high variance, indicating lower consistency compared to our method. In contrast, *ReFEree* demonstrates strong and stable alignment with human judgment, with most data points closely following the ideal correlation line.

5.2 Performance of Segment Level Evaluator

To verify that our segment-level evaluator can accurately identify factual inconsistencies and their types, we analyze prediction accuracy in Table 3. The results show that *ReFEree* achieves an average accuracy of 0.934 (Python) and 0.930 (Java), demonstrating reliable detection of segment-level factual inconsistencies. Furthermore, Table 4 provides qualitative comparisons with other methods, illustrating how *ReFEree* offers explainability by pinpointing where inconsistencies occur and why they are classified as such. Also in Appendix C.4, we briefly demonstrate that *ReFEree* can also be applied to a real-world developer-written summary.

Methods	Python	Java
G-Eval	0.400	0.406
ReFEree (w/ C1)	0.394 (-1.6%)	0.318 (-5.2%)
ReFEree (w/ C2)	0.417 (+4.3%)	0.379 (-6.8%)
ReFEree (w/ C3)	0.419 (+4.8%)	0.391 (+12.7%)
ReFEree (w/ C4)	0.405 (+1.2%)	0.352 (+3.1%)
ReFEree (w/ all)	**0.459** (+14.6%)	**0.480** (+18.1%)

Table 5: Average correlation coefficients (Pearson, Spearman, Kendall's Tau) for *ReFEree* (w/ all), its single-criterion ablations (w/C1-C4), and G-Eval. Percentages indicate relative performance change compared to G-Eval.

5.3 Effectiveness of Considering Four Criteria

To demonstrate that considering all four criteria when evaluating a segment-level code summary is effective, we compare *ReFEree* using all four criteria (w/ all) against single-criterion ablations (w/ C1–C4) in Table 5. We also report performance changes relative to the previous SOTA, G-Eval.

Experimental results show that evaluating all four inconsistency criteria in real-world code summaries produces the most reliable assessment. Among the individual criteria, considering only functionality inconsistency (w/ C3) results in the smallest performance drop, while considering only name inconsistency (w/ C1) causes the largest drop, even falling below G-Eval. These findings suggest that name inconsistencies (C1) are less influential when humans judge the factual consistency of code summaries, whereas functionality (C3) and type inconsistencies (C2) play a more critical role, highlighting their importance in human evaluations. We also analyze how assigning different weights to each criterion during score aggregation affects evaluation performance in Appendix C.5.

5.4 Generalizability Across Different LLMs

To demonstrate that *ReFEree* consistently outperforms existing methods across different LLMs, we conduct a comparative analysis against the G-Eval (previous SOTA). As shown in Table 6, we compare the human correlations when using six various LLMs, respectively, as our segment-level criterion evaluator, four open-LLMs (Llama3, Mistral, Qwen2, and Qwen2.5 coder), and two closed-LLMs (GPT4o-mini and GPT4.1-mini).

As discussed in Section 5.1, *ReFEree* using GPT4.1-mini as the evaluator achieves about a 15% improvement over G-Eval, and similar improvements are observed when using other LLMs. While open small models show lower correlations than closed LLMs, our method still outperforms G-Eval, also when G-Eval incorporates both input code and related information (G-Eval w/ info), demonstrating that our fine-grained, systematic evaluation process achieves consistent performance improvements across different LLM evaluator settings.

Models	Methods	Python	Java
Llama3 (8B)	G-Eval	0.145	0.217
	G-Eval (w/ info)	0.171	0.251
	ReFEree	**0.187**	**0.367**
Mistral (7B)	G-Eval	0.191	0.196
	G-Eval (w/ info)	0.191	0.242
	ReFEree	**0.233**	**0.250**
Qwen2 (7B)	G-Eval	0.221	0.185
	G-Eval (w/ info)	0.243	0.238
	ReFEree	**0.284**	**0.304**
Qwen2.5-Coder (7B)	G-Eval	0.247	0.243
	G-Eval (w/ info)	0.304	0.348
	ReFEree	**0.351**	**0.394**
GPT4o-mini	G-Eval	0.242	0.390
	G-Eval (w/ info)	0.259	0.428
	ReFEree	**0.354**	**0.429**
GPT4.1-mini	G-Eval	0.400	0.406
	G-Eval (w/ info)	0.418	0.463
	ReFEree	**0.459**	**0.480**

Table 6: Generalizability of *ReFEree* Across LLMs. We compare the performance improvements of *ReFEree* across six different LLMs, relative to G-Eval and G-Eval (w/ info) for the average of Pearson, Spearman, and Kendall's Tau correlation coefficients ($p < 0.005$).

Settings	Different Prompts		Different Seeds	
Methods	G-Eval	ReFEree	G-Eval	ReFEree
Python	0.683	**0.733**	0.805	**0.893**
Java	0.400	**0.730**	0.712	**0.901**

Table 7: Stability of *ReFEree*. We compute inter-annotator agreement (IAA) scores across three independent evaluation results for each method (G-Eval and ReFEree) when evaluated under different prompts or different seed settings, for both G-Eval and ReFEree.

5.5 Stability in Evaluation Results

Stability is a critical aspect of any evaluation method, as consistent assessment is essential. From this perspective, Table 7 evaluates the stability of *ReFEree* compared to G-Eval by measuring inter-annotator agreement (IAA) across three repeated evaluations under different seeds and prompts

(prompt variations detailed in Appendix C.6). Under seed variation, *ReFEree* achieves IAA of 0.893 (Python) and 0.901 (Java), demonstrating highly consistent assessments across runs. Under prompt variation, G-Eval shows significant instability with IAA of 0.683 (Python) and 0.400 (Java), whereas *ReFEree* maintains higher IAA of 0.733 (Python) and 0.730 (Java). These results indicate that *ReFEree*'s fine-grained evaluation approach provides more stable and reliable assessments than conventional LLM-based methods.

6 Conclusions

We introduced a novel reference-free and fine-grained method for assessing the factual consistency of real-world code summarization. We also constructed a high-quality human-labeled benchmark to validate the reliability of the proposed method. The comprehensive evaluation results demonstrate that ReFEree achieves high accuracy, reliability, generalizability, and stability, making it effectively applicable to real-world code summarization tasks.

Limitations

ReFEree focuses specifically on evaluating whether LLM-generated summaries contain factually inconsistent information with the code, rather than assessing completeness or overall quality. This design choice is intentional. As highlighted in prior work (Wang et al., 2020; Min et al., 2023), factual consistency is a key factor in ensuring basic reliability and has been repeatedly emphasized as a critical problem that warrants dedicated attention. A factually inconsistent summary can mislead developers regardless of how comprehensive it is. Extending to evaluate completeness or overall summary quality is a promising direction for future work. Also, our approach relies on AST-based static analysis and does not handle dynamic language features (e.g., dynamic dispatch in Python). However, such features accounted for only 0.07% of entities in our benchmark, with no observable performance degradation, suggesting that static syntactic dependencies suffice for reliable factual verification in practice.

Acknowledgmnets

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-01045, 12%; RS-2019-II190421, 13%; IITP-2026-RS-2024-00437633, 12%; RS-2025-25442569, 12%; and IITP-2026-RS-2024-00360227, 13%), National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2025-00521391, 13%) and funded by the Ministry of Education (RS-2025-25433088, 12%), and RCN-NRF project AURoRA (RCN no. 359216, NRF grant RS-2025-03522980, 13%)

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *ArXiv preprint*, abs/2303.08774.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Anthropic. 2025. Claude Code: Agentic Coding Tool. `https://docs.anthropic.com/en/docs/claude-code`. Accessed: 2025.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7957–7977, Bangkok, Thailand. Association for Computational Linguistics.

YunSeok Choi, JinYeong Bak, CheolWon Na, and Jee-Hyong Lee. 2021. Learning sequential and structural information for source code summarization. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2842–2851, Online. Association for Computational Linguistics.

YunSeok Choi, Hyojun Kim, and Jee-Hyong Lee. 2023. BLOCSUM: Block scope-based source code summarization via shared block representation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 11427–11441, Toronto, Canada. Association for Computational Linguistics.

Michael Desmond, Michael Muller, Zahra Ashktorab, Casey Dugan, Evelyn Duesterwald, Kristina Brimijoin, Catherine Finegan-Dollak, Michelle Brachman, Aabhas Sharma, Narendra Nath Joshi, and 1 others. 2021. Increasing the speed and accuracy of data labeling through an ai assisted interface. In *Proceedings of the 26th International Conference on Intelligent User Interfaces*, pages 392–401.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445, Torino, Italia. ELRA and ICCL.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *ArXiv preprint*, abs/2308.01861.

GitHub. 2021. GitHub Copilot: Your AI Pair Programmer. `https://github.com/features/copilot`. Accessed: 2025.

Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304. Automated Software Evolution.

Zhen Ming Jiang and Ahmed E. Hassan. 2006. Examining the evolution of code comments in postgresql. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 179–180, New York, NY, USA. Association for Computing Machinery.

Sungmin Kang, Louis Milliken, and Shin Yoo. 2024. Identifying inaccurate descriptions in llm-generated code comments via test execution. *ArXiv preprint*, abs/2406.14836.

Hannah Kim, Kushan Mitra, Rafael Li Chen, Sajjadur Rahman, and Dan Zhang. 2024. MEGAnno+: A human-LLM collaborative annotation system. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, pages 168–176, St. Julians, Malta. Association for Computational Linguistics.

Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, and 1 others. 2024. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. *ArXiv preprint*, abs/2405.19856.

Junyi Li, Xiaoxue Cheng, Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. 2023. HaluEval: A large-scale hallucination evaluation benchmark for large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6449–6464, Singapore. Association for Computational Linguistics.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024a. Exploring and evaluating hallucinations in llm-powered code generation. *ArXiv preprint*, abs/2404.00971.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024b. Repobench: Benchmarking repository-level code auto-completion systems.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: NLG evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522, Singapore. Association for Computational Linguistics.

Vadim Lomshakov, Andrey Podivilov, Sergey Savin, Oleg Baryshnikov, Alena Lisevych, and Sergey Nikolenko. 2024. ProConSuL: Project context for code summarization with LLMs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 866–880, Miami, Florida, US. Association for Computational Linguistics.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages

436–464, Miami, Florida, USA. Association for Computational Linguistics.

Kishan Maharaj, Vitobha Munigala, Srikanth G. Tamilselvam, Prince Kumar, Sayandeep Sen, Palani Kodeswaran, Abhijit Mishra, and Pushpak Bhattacharyya. 2025. ETF: An entity tracing framework for hallucination detection in code summaries. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30639–30652, Vienna, Austria. Association for Computational Linguistics.

Antonio Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, and Gabriele Bavota. 2024. Evaluating code summarization techniques: A new metric and an empirical characterization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2023. FActScore: Fine-grained atomic evaluation of factual precision in long form text generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12076–12100, Singapore. Association for Computational Linguistics.

Atharva Naik, Marcus Alenius, Daniel Fried, and Carolyn Rose. 2024. Crscore: Grounding automated evaluation of code review comments in code claims and smells. *ArXiv preprint*, abs/2409.19801.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Olga Russakovsky, Li-Jia Li, and Fei-Fei Li. 2015. Best of both worlds: Human-machine collaboration for object annotation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 2121–2131. IEEE Computer Society.

Hwanjun Song, Hang Su, Igor Shalyminov, Jason Cai, and Saab Mansour. 2024. Finesure: Fine-grained summarization evaluation using llms. *ArXiv preprint*, abs/2407.00908.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2024. Source code summarization in the era of large language models. *ArXiv preprint*, abs/2407.07959.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25300–25308.

Alex Wang, Kyunghyun Cho, and Mike Lewis. 2020. Asking and answering questions to evaluate the factual consistency of summaries. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5008–5020, Online. Association for Computational Linguistics.

Jiaan Wang, Yunlong Liang, Fandong Meng, Zengkui Sun, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. 2023a. Is ChatGPT a good NLG evaluator? a preliminary study. In *Proceedings of the 4th New Frontiers in Summarization Workshop*, pages 1–11, Singapore. Association for Computational Linguistics.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023b. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290, Singapore. Association for Computational Linguistics.

Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64.

Yang Wu, Yao Wan, Zhaoyang Chu, Wenting Zhao, Ye Liu, Hongyu Zhang, Xuanhua Shi, and Philip S Yu. 2024. Can large language models serve as evaluators for code summarization? *ArXiv preprint*, abs/2412.01333.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Zheng Zhang, Zheng Ning, Chenliang Xu, Yapeng Tian, and Toby Jia-Jun Li. 2023. Peanut: A human-ai collaborative tool for annotating audio-visual data. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–18.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao,

and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

A Additional Details About ReFEree

A.1 More details of Factual Inconsistency Criteria Analysis

In Section 3.1, we conduct an analysis to accurately identify factual inconsistency patterns that occur in real-world code summaries. To this end, we generate a total of 300 code summaries using three different models: Qwen2.5-Coder (7B), CodeLlama (7B), and GPT-4 (gpt-4o-mini). For summary generation, we use identical hyperparameters across all models: temperature = 0.6, top-p = 0.9, and max new tokens = 256. The prompt template used for generation is shown below, where **{lang}** corresponds to either Python or Java.

Code summarization prompt

You are an expert in understanding **{lang}** code. As a **{lang}** expert, please generate a detailed and informative summary of the following python function. The generated summary should describe the implementation details of any methods that the function use it.
Provide ONLY the summary texts. Do not include any other codes, or notes.
CODE: {input_code}
SUMMARY:

Three human annotators independently review each generated summary and manually annotate any parts of factual inconsistencies along with their comments. To ensure annotation reliability, we only count an inconsistency when at least two out of three annotators identify the same comment in the same segment. Table 8 presents the frequency of each factual inconsistency type across different models. Our analysis reveals that functionality inconsistency (C3) and context irrelevant (C4) are the most frequently occurring error types across all models, followed by name inconsistency (C1) and type inconsistency (C2). The remaining 3% of cases, categorized as Others, mostly consist of repeated or redundant context within the summary. Additionally, Table 9 provides representative examples of each factual inconsistency criterion identified through our human analysis.

Pattern	C1	C2	C3	C4	Others	Sum
Qwen2.5-Coder	14	24	51	39	4	132
CodeLlama	23	24	55	52	5	159
GPT4	15	10	28	36	4	93
Sum	52	58	134	127	13	384
Distribution	14%	15%	35%	33%	3%	100%

Table 8: The frequency of factual inconsistency patterns for each model, and the overall distribution of these patterns. The four patterns (C1-C4) are followed in Section 3.1.

A.2 Code-Related Information Searching

This section provides a detailed description of the code-related information searching method introduced in Section 3.2.

Step1: Project context graph construction When constructing a project context graph, we build a heterogeneous directed acyclic graph (DAG) of code entities and their dependency relationships based on abstract syntax trees (AST) (Zhang et al., 2019) and static program analysis techniques. Each node stores metadata such as the entity's AST type, code context, and docstring. The edges represent the flow of variables, indicating where they originate and where they are propagated. To capture variable relationships in a more structured way, we apply five relation types to the edges, as defined by Cheng et al. (2024):

- **Assign** relation is a one-to-one correspondence in an assignment statement, which controls variable creation and mutation
- **As** relation is from with or except statements and similar to the *assigns* relation.
- **Refers** relation that represents a reference to an existing variable or its attribute.
- **Typeof** relation indicates the data type of the (return) value of a variable or function.
- **Inherits** relation is an implicit data dependency relation since a subclass inherits all the class members of its base classes.

Our DFG is a heterogeneous directed acyclic graph $G = \{(h, r, t)|h, t \in E, r \in R\}$, where E denotes the entity set, R denotes the type-sensitive relation set, and the triplet (h, r, t) represents the head entity h pointing to the tail entity t with the relation r.

Step2: Code-related information selection In the next step, we extract relevant node information from the project context graph based on the input code's dependency relationships.

Criteria	Input Code	Generated Summary	Inconsistency Reason

C1. Name Inconsistency	def where_filters(request, database, datasette):
	// Some code . . .
	extra_wheres_for_ui = [
	{
	"text": text,
	"remove_url": path_with_removed_args (request, "_where": text),
	} for text in request.getlist("_where")]
	// Some code . . .
	return inner
		[...] This removal URL is constructed using the helper function 'path_with_removed_args', which builds a new URL path by excluding query parameters identified by the 'should_exclude' function. [...]	Generated summary describes that the sentence explaining the functionality states that the 'should_exclude' function inside 'path_with_removed_args' identifies the query parameters to be excluded. **However, in the actual code, the function is defined as 'should_remove', not 'should_exclude'**. Considering the related dependencies information within the input code, this discrepancy falls under **name inconsistency**.

C2. Type Inconsistency	def scan(self, package=None, ... , **kw,):
	package = self.maybe_dotted(package)
	if package is None:
	package = return_package()
	ctorkw = 'config': self
	ctorkw.update(kw)
	// Some code . . .
		[...] The 'return_package' function returns the name of the calling module's package as a string. [...]	When examining the actual functionality of 'return_package', we find that it actually **returns the calling package as a 'sys.modules' object**. This discrepancy falls under **type inconsistency**.

C3. Functionality Inconsistency	def _is_asgi_app(app):
	app_args = inspect.getfullargspec(app)
	num_app_args = len(app_args)
	// Some code . . .
	if app_args[0] in 'cls', 'self':
	num_app_args -= 1
	is_asgi = num_app_args == 3
	return is_asgi
		[...] It uses the 'inspect' module to retrieve the full argument specification of the function or method passed as 'app'. [...]	This summary insufficiently and inaccurately describes the dependency function's behavior. The actual functionality of **'inspect.getfullargspec()' does not merely retrieve the full arguments.** This function initializes the app parameter and then returns the full set of argument values.

C4. Context Irrelevant	// Some code . . .
	if settings.USE_TZ and is_naive(dt):
	return make_aware(dt, timezone=timezone.utc)
	// Some code . . .
	return dt
		[...] UTC (Coordinated Universal Time) timezone is the primary time standard by which the world regulates clocks and time.	The generated summary **includes an additional explanation about the UTC timezone, which is unrelated to the function's actual behavior**. This constitutes irrelevant context in the summary.

Table 9: Examples and explanations about four major patterns of factual inconsistency detected by human annotators. Factual inconsistencies between the input code and the generated summary are highlighted in red.

In this paper, information related to each entity in the input code is selected through the following steps for efficiency. **(1)** We perform a depth-first search (DFS) on the project context graph, starting from each entity, to traverse all related nodes connected via dependency edges. **(2)** If any of the related nodes for a given entity are `import_statement` (`import_declaration` for Java language, they are transformed into the form `(module, name)` or `(module, name.attr)`. For example, "`from utils.collection_utils import key_prefix`" is converted to "`(utils.collection_utils, key_prefix)`". **(3)** If the `module` path exists within the project directory, the given entity is considered to have a **cross-file dependency**. In this case, we perform an exact match search that precisely corresponds to the `name` or `name.attr` within that `module` file. If the searched corresponding node type is `class_definition`, `function_definition`, or `assignment`, we select the corresponding node context (`class_declaration`, `method_declaration`, or `field_declaration` for Java language.). **(4)** If the `module` path does not exist within the project directory and is imported from external sources such as Python built-in or third-party libraries, the given entity is considered to have an **external dependency**. In such cases, we select the corresponding description for the given *name* or *name.attr* from a predefined external API documentation, collected from ODEX (Wang et al., 2023b), DS-1000 (Lai et al., 2023), and the official Python and Java documentation site[3]. **(5)** Finally, if none of the related nodes are of the `import_statement` type, it indicates that the dependency information for the given node is located within the same file as the input code (**internal-file dependency**). In this case, if any related nodes are of type `class_definition`, `function_definition`, or `assignment` (`class_declaration`, `method_declaration`, or `field_declaration` for Java language.), we select the corresponding node context. **(6)**

[3]Python : https://docs.python.org/3/, Java: https://docs.oracle.com/en/java/

The information obtained through steps (1–5) is prepended to the input code in the format "# {name} # {content}". If the information corresponds to a class, the content contains the class docstring. For functions or variables, the content includes the code body.

A.3 Segment-Level Evaluator

At the sentence-level factual consistency evaluation stage, **we focus on defining sentence-level judgments in terms of an objectively verifiable "presence or absence of error"**, designing a clear binary prompt in which a sentence is assigned a score of 0 if any factual inconsistency is detected with respect to a given criterion.

The prompt for segment-level factual inconsistency criterion evaluation is described below. The {criterion} and {explanation} fields contain the name (e.g., Name Inconsistency) and definition described in Table 1. This prompt is input into the model, which outputs either 1 or 0 at segment and criteria level.

Segment-level criterion evaluation prompt

[System Prompt]
You will be given one summary text written for a source code. Your task is to evaluate the summary from {criterion} aspect. Please make sure you read and understand these instructions carefully. Please keep this document open while reviewing, and refer to it as needed.

Evaluation Criteria:
{criterion} (1 or 0) – {explanation}

Evaluation Steps:
1. Read the CODE carefully and understand its main intent.
2. Read the code summary text and check if it accurately describes the code.
3. Evaluate whether {criterion} exists, where "1" means "{criterion} does not exist" and "0" means "{criterion} exists" based on the Evaluation Criteria.

[User Prompt]
CODE:
(Related Information) {related_information}
(Input Code) {input_code}
SUMMARY TEXT: {segment}
SCORE (score only):

B Additional Details About Experimental Setups

B.1 More Details About Benchmark Construction Process

Summary Generation Prompt: The goal of our benchmark is *(1)* to evaluate whether ReFEree can distinguish the four factual consistency criteria (C1-C4) in a manner similar to human judgement, and *(2)* to assess whether ReFEree's final scores correlate well with human ratings.

To properly evaluate these capabilities, the benchmark must contain a sufficiently balanced number of gold labels across all four criteria. However, when using only naturally generated summaries as-is, we observe two issues (as shown in the Table 8; The overall frequency of factual inconsistencies is relatively low, and there exists a significant label imbalance across the four criteria. This imbalance makes it difficult to reliably measure criterion-level evaluation ability. Therefore, to ensure sufficient coverage for all evaluation dimensions, we intentionally incorporated inconsistency patterns into the prompt when generating hallucinated summaries.

Importantly, these inconsistency patterns are not arbitrarily designed. In Section 3.1, we analyzed naturally generated summaries from three LLMs and identified four representative, naturally occurring error patterns. These empirically observed patterns were then reflected in the construction of hallucinated summaries. In other words, the inconsistency types included in our benchmark are grounded in real-world error distributions rather than being artificially invented rubric-specific artifacts.

Below is the full prompt for generating code summaries with intentional factual inconsistencies. We use the ChatGPT model (`chatgpt-4o-latest`), with the following hyperparameter settings: temperature = `0.6`, top-p = `0.9`, and max new tokens = `256`. The total cost of generating hallucinated summaries for 2,075 samples amounts to approximately $250. **`{lang}`** corresponds to either Python or Java.

Hallucinated summary generation prompt

[System Prompt]
You are an expert in understanding **`{lang}`** code. As a **`{lang}`** expert, please generate a detailed and informative hallucinated summary of the following **`{lang}`** code. The hallucinated

summary contains context that sounds plausible but does not accurately reflect the actual implementation or behavior of the code.
You SHOULD generate a summary that includes at least one (one or more) hallucinated sentence corresponding explicitly to one of the following five factual inconsistency cases, respectively.:
1. The name of a function, class, or variable mentioned in the text does not match the actual identifier used in the code.
<Demonstration>
code: **{code}**
Hallucinated Summary: **{summary}**
2. The described return type or variable type in the text is inconsistent with the actual type used or inferred in the code.
<Demonstration>
3. The described functionality or purpose of the code in the text does not accurately reflect what the Python code actually implements.
<Demonstration>
4. The described text contains content that is unnecessary or unrelated to the input code.
<Demonstration>
You should try your best to make the hallucinated summary. Provide ONLY the summary texts. Do not include any other codes or notes.

[User Prompt]
CODE:
(Related Information) {related_information}
(Input Code) {input_code}
HALLUCINATED SUMMARY:

Factual Consistency Labeling Process: As many prior studies (Russakovsky et al., 2015; Desmond et al., 2021; Zhang et al., 2023; Kim et al., 2024) have noted, the primary advantage of human–AI collaborative labeling systems is that AI and humans can complement each other's weaknesses, thereby improving annotation quality. Human annotators use their expertise to verify and correct hallucinations and factual errors that may occur in LLM-generated labels, while LLM-based labelers rapidly propose initial candidate labels for large-scale data, reducing the excessive annotation time and subjectivity that arise when relying solely on humans.

First, three LLMs with different roles determine the candidate label through majority voting. If all three LLMs assign different scores during summary-level labeling (since there are five possible labels), the score closest to the mean is selected as the candidate label. The two greyboxes below show the full prompts used to determine candidate labels for summary-level and segment-level annotations, respectively. We use ChatGPT (`chatgpt-4o-latest`) with the following hyperparameter settings: temperature = `0.1`, top-p = `0.9`, and max new tokens = `16`. The total cost of predicting labels for 2,055 samples amounts to approximately $300 at the summary level and $800 at the segment level. **{lang}** corresponds to either Python or Java and each LLM is assigned a distinct {role}: Code Editor, Code Reviewer, or Original Code Author.

Summary-level labeling prompt

[System Prompt]
As a {role}, your task is to rate the factual consistency of a summarized text generated from a **{lang}** code and related information.
Factual Consistency: Guarantee that the summary remains consistent with the CODE and RELATED INFORMATION, accurately capturing its primary functionality and logic without adding any unrelated content.
Please refer to the CODE, RELATED INFORMATION, and the SUMMARY, and then assign a factual consistency score based on the following grading rubric.

SCORE RUBRIC:
- 5: Highly Consistent : All information in the generated content can be verified in the CODE and RELATED INFORMATION.
- 4: Very Consistent : Most information in the generated content can be verified in the CODE and RELATED INFORMATION, with one minor item that wouldn't negatively impact the reader's understanding.
- 3: Moderately Consistent : More than one piece of information in the generated content cannot be verified in the CODE and RELATED INFORMATION, but none of these inaccuracies would negatively impact the reader's understanding.
- 2: Somewhat Inconsistent : One or more pieces of information in the generated content are factually inaccurate and cannot be verified in the CODE and RELATED INFORMATION, some, or all of which would negatively impact the reader's understanding.

- 1: Highly Inconsistent : Most or all of the information in the generated content is inaccurate, cannot be verified in the CODE and RELATED INFORMATION, and would negatively impact the reader's understanding.

Generate ONLY the score. Do not include any other codes, or notes.

[User Prompt]
CODE:
(Related Information) {related_information}
(Input Code) {input_code}
SUMMARY: {summary}
SCORE:

Segment-level labeling prompt

As a {role}, you ensure the factual consistency with the **`{lang}`** code and the given sentence, which is a part of the code summary.
The given text should be precise and only include verifiable information that is explicitly stated in the source code, so do not make any assumptions or derive any thoughts.

Given the CODE, RELATED INFORMATION, and the SUMMARY TEXT, which is the part of the summary, your objective is to evaluate whether the sentence contains {criterion} that {explanation}. If the sentence strictly corresponds to a {criterion}, you should output 0; otherwise, output 1.

[User Prompt]
CODE:
(Related Information) {related_information}
(Input Code) {input_code}
SUMMARY TEXT: {segment}
SCORE:

Then, three human annotators subsequently post-edit the candidate label to determine the final label. They are instructed to retain the label if they agree with the candidate label, or to revise it if they do not. The final label is determined according to the following three rules: (1) If at least two annotators agree with the candidate label, the label is retained. (2) If at least two annotators revise to the same label, the label is edited accordingly. (3) For summary-level labeling: If all annotators assign different labels, the final label is determined through further discussion among the three annotators. Figure 5 shows the platform used for the human annotation process.

Summary-Level				
Label	**Python**		**Java**	
1 (Highly Inconsistent)	16		7	
2 (Somewhat Inconsistent)	829		54	
3 (Moderately Consistent)	435		74	
4 (Very Consistent)	488		87	
5 (Highly Consistent)	57		8	
Total	**1,825**		**230**	
Segment-Level				
Label	**1**	**0**	**1**	**0**
C1 (Name Inconsistent)	3,815	6,600	983	406
C2 (Type Inconsistent)	2,844	7,571	1,131	258
C3 (Functionality Inconsistent)	3,975	6,440	700	689
C4 (Context Irrelevant	4,690	5,725	635	754
Total	**10,415**		**1,389**	

Table 10: Label distribution of our benchmark. Summary-level labels range from 1 (highly inconsistent) to 5 (highly consistent). Segment-level labels indicate whether each criterion (C1–C4) is CORRESPOND (1) or NOT CORRESPOND (0).

Dataset Statistics and Examples: Table 10 presents the label distribution of our benchmark. The benchmark consists of 2,055 code summaries (1,825 Python and 230 Java) annotated at both summary and segment levels. At the summary level, each sample is labeled on a 5-point scale ranging from 1 (highly inconsistent) to 5 (highly consistent). At the segment level, the benchmark contains 11,804 annotations (10,415 Python and 1,389 Java), where each sentence is labeled as CORRESPOND (1) or NOT CORRESPOND (0) for each of the four factual inconsistency criteria (C1–C4). Table 19 provides some examples of our benchmark.

Regarding Potential Risks of Systemic Bias During Annotation: To mitigate potential annotator bias from LLM-provided candidate scores, our labeling pipeline treats model scores as non-binding references rather than ground truth. Annotators are explicitly instructed that scores may be freely overridden, and final labels are determined by majority voting across three LLMs and three human annotators, ensuring that neither human nor model judgment dominates.

To empirically validate this design, we compare labels produced under our *Human-AI* collaboration system against labels collected entirely by humans from scratch (*human-only*). The two settings yield an agreement rate of 89.5%, and among

the 10.5% of mismatched cases, the mean absolute score difference is only 0.105 - with all disagreements falling within ± 1 point and no case exceeding a 2-point gap. Label correlation is consistently high (Pearson = 0.9401, Spearman = 0.9499, Kendall = 0.9242), confirming that the *Human-AI* approach preserves annotation quality while substantially reducing cost and time.

B.2 Details About Baselines

We select 13 commonly used evaluation methods, 8 reference-based and 5 reference-free methods, from code summarization tasks to compare with our proposed method.

1. Reference-based methods: We use the English descriptions as reference summaries.

ROUGE (Lin, 2004) ROUGE measures the overlap of n-grams between the generated output and reference summaries. In this paper, we use ROUGE-1/2/L f1 score for baselines.

BLEU (Papineni et al., 2002) measures the n-gram precision between the generated text and references with an additional brevity penalty to discourage short outputs.

BERTScore (Zhang et al., 2020) BERTScore computes similarity between generated and reference texts using contextual embeddings from pre-trained BERT models. It captures semantic similarity better than traditional lexical metrics.

METEOR (Banerjee and Lavie, 2005) METEOR, a recall-oriented metric, evaluates how well the model captures reference content by matching words between candidate and reference sentences and computing the harmonic mean of precision and recall.

Sentencebert (Reimers and Gurevych, 2019) SentenceBERT encodes sentences into dense vector representations to compute semantic similarity via cosine distance or Euclidean distance.

2. Reference-free methods: We evaluate the overall factual consistency between the input code and the entire summary with 5 baselines using the same LLM, GPT-4.1-mini (`gpt-4.1-mini-2025-04-14`), and the hyperparameters are set as follows: temperature = 0.1, top-p = 0.9, top-k = 50, and max new tokens = 4. Since LLM-Judge, G-Eval, and FactScore are originally designed as evaluation methods for the NLP domain, we modify their prompts to ensure applicability to the code domain.

SIDE (Mastropaolo et al., 2024) SIDE evaluates the semantic fitness of code summaries by using contrastive learning to distinguish between characteristics of suitable and unsuitable summaries for a given code. We used the pre-trained SIDE model from GitHub [4].

LLM-Judge (Zheng et al., 2023) This paper was the first to propose the LLM-as-a-Judge framework. We utilize the default prompt for single-answer grading from this paper, inputting the code and summary to evaluate consistency. The prompt is shown below.

LLM-judge prompt

[System Prompt]
Please act as an impartial judge and evaluate the quality of the response provided by an AI assistant to the user question displayed below.
Your evaluation should consider the factually consistency that the summary remains consistent with the original code, accurately capturing its primary functionality and logic without adding any unrelated content.
Please rate the response on a scale of 1 to 5.

[User Prompt]
CODE: {input_code}
SUMMARY TEXT: {summary}
SCORE (score only):

G-Eval (Liu et al., 2023) This method is a framework that utilizes Large Language Models (LLMs) with Chain-of-Thought (CoT) and a form-filling paradigm to assess the quality of Natural Language Generation (NLG) outputs. We adapt the prompt to be suitable for code summary evaluation, as shown below.

G-eval prompt

[System Prompt]
You will be given one summary written for a source code. Your task is to rate the summary from factually consistency aspect. Please make sure you read and understand these instructions carefully. Please keep this document open while reviewing, and refer to it as needed.
Evaluation Criteria:
Consistency (1-5) - Guarantee that the summary remains consistent with the original code, accurately capturing its primary functionality

[4] https://github.com/antonio-mastropaolo/code-summarization-metric

and logic without adding any unrelated content.

Evaluation Steps:
1. Read the CODE carefully and understand its main intent.
2. Read the code summary and check if it accurately describe the code.
3. Assign a score for factually consistency on a scale of 1 to 5, where 1 is the lowest and 5 is the highest based on the Evaluation Criteria.

[User Prompt]
CODE: {input_code}
SUMMARY TEXT: {summary}
SCORE (score only):

Factscore (Min et al., 2023) This method is a fine-grained method proposed in the NLP fields that breaks a generation into a series of atomic facts and computes the percentage of atomic facts supported by a reliable knowledge source. For use in code summary evaluation, we break down the code summary into a series of atomic facts and compute the percentage of consistency with the code. We first generate these atomic facts using the prompt: "`Please breakdown the following sentence into independent facts.`" Subsequently, each atomic fact is evaluated using the prompt: "`Answer the question based on the given context. Context: {code} Input: {summary} True or False? Output:`". The input code is provided as the knowledge source instead of general background knowledge.

CODERPE (Wu et al., 2024) This method uses a Role-Player system prompt designed to quantify the quality of generated code summaries. We prompt an LLM agent to perform a role, code reviewer. This task involves evaluating the quality of generated code summaries, specifically along the dimension of consistency. The prompt is shown below.

CODERPE prompt

[System Prompt]
As a code reviewer, you ensure the factually consistency of the code summary.

You will be given one summary written for a source code. Your task is to rate the summary from factually consistency aspect. Please make sure you read and understand these instructions carefully. Please keep this document open while reviewing, and refer to it as needed.

Evaluation Criteria:
Consistency (1-5) - Guarantee that the summary remains consistent with the original code, accurately capturing its primary functionality and logic without adding any unrelated content.

Evaluation Steps:
1. Read the CODE carefully and understand its main intent.
2. Read the code summary and check if it accurately describe the code.
3. Assign a score for factually consistency on a scale of 1 to 5, where 1 is the lowest and 5 is the highest based on the Evaluation Criteria.

[User Prompt]
CODE: {input_code}
SUMMARY TEXT: {summary}
SCORE (score only):

B.3 Implementation Details

Our method supports various LLMs, including both closed-source and open-source models, as segment-level criterion evaluators. In our main experiments, we use OpenAI's GPT-4.1-mini to ensure a fair comparison with existing baselines under the same setting. All evaluations are conducted with consistent hyperparameters: temperature = `0.1`, top-p = `0.9`, top-k = `50`, and max new tokens = `4`.

C Additional Experimental Results

C.1 Information Searching Ablations

We additionally presented experimental results comparing evaluations using 0, 1, and 2-hop context in Python benchmarks. As shown in Table 11, the 1-hop setting achieves the highest average correlation (0.459), outperforming both 0-hop (0.404) and 2-hop (0.448). While using 2-hop context did improve correlation compared to 0-hop, it still underperformed the 1-hop setting. The key insights derived from this analysis are as follows:
1) Our analysis shows that most factual inconsistencies can be correctly determined based on information from directly invoked entities (depth-1).
2) When expanding retrieval to 2-hop or deeper, the additional information such as transitive dependencies, internal implementation details or indirect call-chain information typically includes. However, our experimental results show that this additional

Context setting	r_p	r_s	τ	Average
0-hop (w/o info)	0.432	0.432	0.349	0.404
1-hop (ours)	**0.497**	**0.489**	**0.390**	**0.459**
2-hop	0.491	0.474	0.377	0.448

Table 11: Comparison of correlation coefficients across different search depths for code-related information.

information is rarely mentioned in actual code summaries and is not typically considered in human factuality judgments. Instead, it tends to introduce irrelevant context, which can make the evaluation less stable and less accurate.

C.2 Time and computational costs

We compare the per-sample execution time and cost between baseline methods and our approach, with results presented in Table 12. Most reference-based methods incur no API cost, making them computationally efficient. However, their low correlation with human judgment makes them unsuitable for evaluating long, descriptive project-level code summaries. Our method, ReFEree, requires longer execution time per sample compared to simpler baselines due to its fine-grained evaluation process. However, at the evaluation stage, ensuring reliability and accuracy is more critical than marginal gains in efficiency. For instance, G-Eval is fast (approximately 1.11 seconds per sample) but shows limited correlation with human judgment. FactScore adopts a similar fine-grained framework and requires approximately 9.18 seconds per sample, yet achieves lower correlation (0.391 in Python, 0.355 in Java). In contrast, ReFEree achieves the highest correlation (0.459 in Python, 0.480 in Java) within a comparable time range, demonstrating a favorable trade-off between computational cost and evaluation quality.

C.3 Examples of *ReFEree*

The results of evaluating factual consistency through our method are shown in Tables 20 and Table 21. For both examples, existing methods such as ROUGE-L, BERTScore, and SIDE produce results that differ significantly from human labels in their evaluation of the factual consistency between the code and the generated summary. However, the *ReFEree* method outputs consistency scores that are closer to human label evaluations. Additionally, our method can explain which parts of the generated summary and what types of inconsistencies occur.

Methods	Time/s	Cost/s
ROUGE-1	0.00	0.0000
ROUGE-2	0.00	0.0000
ROUGE-L	0.00	0.0000
BLEU	1.12	0.0000
METEOR	0.02	0.0000
BERTScore	0.03	0.0000
SBCS	0.02	0.0000
SBED	0.03	0.0000
SIDE	0.08	0.0000
LLM-judge	0.34	0.0002
G-Eval	1.11	0.0002
Factscore	9.18	0.0058
CODERPE	0.53	0.0002
ReFEree	10.24	0.0042

Table 12: Comparison of inference time (seconds) and cost ($) per sample across evaluation methods. Average time and cost are computed over 2,055 code summary evaluations in our benchmark.

C.4 Applicable to Human-Written Summaries

Our method focuses on accurately evaluating factual inconsistencies in LLM-generated code summaries. However, ReFEree is not limited to LLM outputs. Our method can also evaluate summaries written by real developers. We additionally collected 183 human-written docstrings from the Deveval dataset along with their corresponding human-annotated factual consistency scores. Based on this data, we performed a comparative analysis against existing baselines.

As shown in Table 13, human-written summaries rarely contain factual errors. Consequently, the human scores are highly skewed. This results in extremely low label variance, making meaningful correlation difficult to obtain. Nevertheless, ReFEree still achieved higher correlation than the baseline methods. This result suggests that ReFEree can meaningfully evaluate real human-written documentation and quantitatively capture the intuitive observation that high-quality human summaries tend to exhibit greater factual consistency.

C.5 Criterion Weight Configuration

In our scoring aggregation, we assign equal weights to each criterion (C1–C4) by default. This design choice does not imply that all criteria are

Method	Score	r_p	r_s	τ	Average
Human (1–5)	4.938	x	x	x	x
ROUGE-L (0–1)	0.208	0.006	0.047	0.039	0.030
BLEU (0–1)	0.030	0.062	0.061	0.054	0.059
G-Eval (1–5)	3.043	0.013	0.003	0.003	0.006
ReFEree (0–1)	**0.938**	**0.326**	**0.298**	**0.287**	**0.304**

Table 13: Evaluation results on 183 human-written docstrings from the DevEval dataset.

Weight	r_p	r_s	τ	Average
1 : 1 : 1 : 1	0.497	0.489	0.390	0.459
0.6 : 1.2 : 1.2 : 1.0	**0.498**	**0.493**	**0.394**	**0.462**

Table 14: Comparison of correlation scores using python benchmark between equal weighting and weighted aggregation for criteria (C1: C2: C3: C4).

equally important, but rather reflects a configuration that treats all criteria evenly without introducing additional hyperparameters. To explore whether weighted aggregation improves performance, we compute the final score by assigning different weights to each criterion based on its individual correlation with human judgment. As shown in Table 14, the weighted configuration (0.6:1.2:1.2:1.0 for C1:C2:C3:C4) yields a slight improvement in average correlation compared to equal weighting (0.462 vs. 0.459). This can be viewed as an optional variant that offers marginal performance gains at the cost of increased metric complexity.

C.6 Prompt sensitivity

In Section 5.5, we experiment with the stability of the G-Eval and ReFEree methods under different prompt settings. We designed the following prompt variations for both the G-Eval and ReFEree prompts. The system prompts for both G-Eval and ReFEree consist of three elements: instruction, evaluation criteria, and evaluation steps. We conduct experiments by constructing three combinations of these elements: instruction + evaluation criteria (ver1), instruction + evaluation steps (ver2), and instruction + evaluation criteria + evaluation steps (ver3). Zero-shot LLMs are sensitive to the template used, meaning that changes in the tokens within the template can significantly impact performance. Table 15 shows that our evaluation method achieves consistent performance even when the prompt is modified.

	Methods	r_p	r_s	τ	Average
Python	ver1	0.489	0.470	0.375	0.445
	ver2	0.492	0.480	0.385	0.452
	ver3*	**0.497**	**0.489**	**0.390**	**0.459**
Java	ver1	0.504	0.488	0.414	0.469
	ver2	0.494	0.471	0.397	0.454
	ver3*	**0.515**	**0.502**	**0.423**	**0.480**

Table 15: Results of coefficient with varying instruction templates in ReFEree. **ver3*** is the final prompt used in our evaluation method.

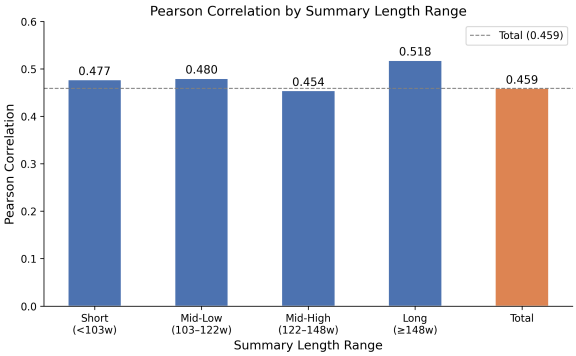


Figure 4: Stability of ReFEree across summary lengths

C.7 Stability Across Summary Lengths

We additionally analyzed how ReFEree's evaluation results vary with summary length. To verify whether summary length affects correlation with human judgments, we divided the 1,825 Python benchmark samples into four groups based on summary length and measured the correlation between ReFEree scores and human labels within each group.

The Figure 4 show that the overall correlation (0.459) remains consistently stable across all groups, ranging from approximately 0.45 to 0.52. This indicates that ReFEree maintains stable alignment with human judgment regardless of summary length. Since ReFEree measures the proportion of factually consistent content within a summary, the same error may result in different final scores depending on the summary length. However, this behavior is consistent with how humans evaluate summaries and does not introduce instability. The results show that this effect is not overly sensitive and remains well aligned with human judgments.

C.8 Failure Analysis

This section describes ReFEree's failure analysis. **Error rate tends to increase as the number of dependency relations in the code grows. (Table 16)**

In particular, when the number of dependencies exceeds 11, the error rates for C3 and C4 rise noticeably. This trend suggests that as dependency information becomes more complex, making it harder to distinguish relevant signals from noise.

#Dependencies	C1	C2	C3	C4	Average
0	3.8%	2.9%	2.6%	4.1%	3.4%
1–2	3.1%	2.7%	3.8%	4.9%	3.6%
3–5	5.2%	3.6%	4.3%	6.2%	4.8%
6–10	5.7%	2.9%	5.1%	6.3%	5.0%
11+	4.7%	2.4%	10.6%	9.4%	6.8%

Table 16: Error rate tends to increase as the number of dependency relations in the code grows.

Error rates differ by dependency type. (Table 17) Internal dependencies (within the same file) show the lowest error rates, while cross-file and external API dependencies are higher. This is likely because internal dependencies rely on localized context, whereas cross-file and external cases require additional reasoning over dispersed information, increasing difficulty.

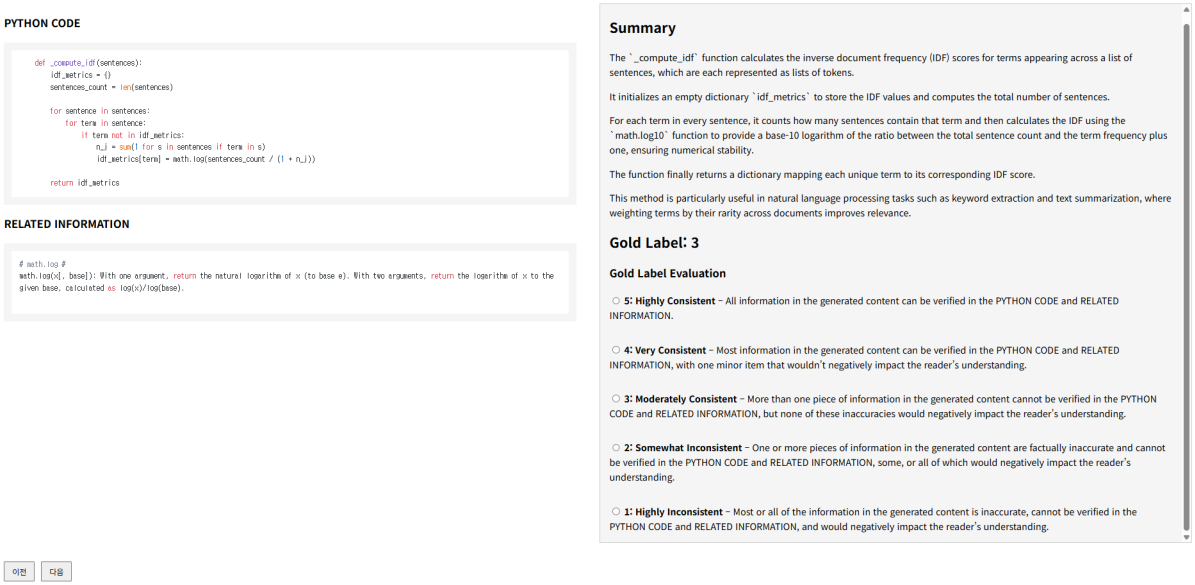
Dependency Type	C1	C2	C3	C4	Average
Internal	5.8%	3.4%	5.1%	5.8%	**5.0%**
Cross	8.7%	2.4%	6.7%	9.1%	**6.7%**
External	6.9%	4.9%	7.2%	8.0%	**6.7%**

Table 17: Error Rates (%) across different dependency type

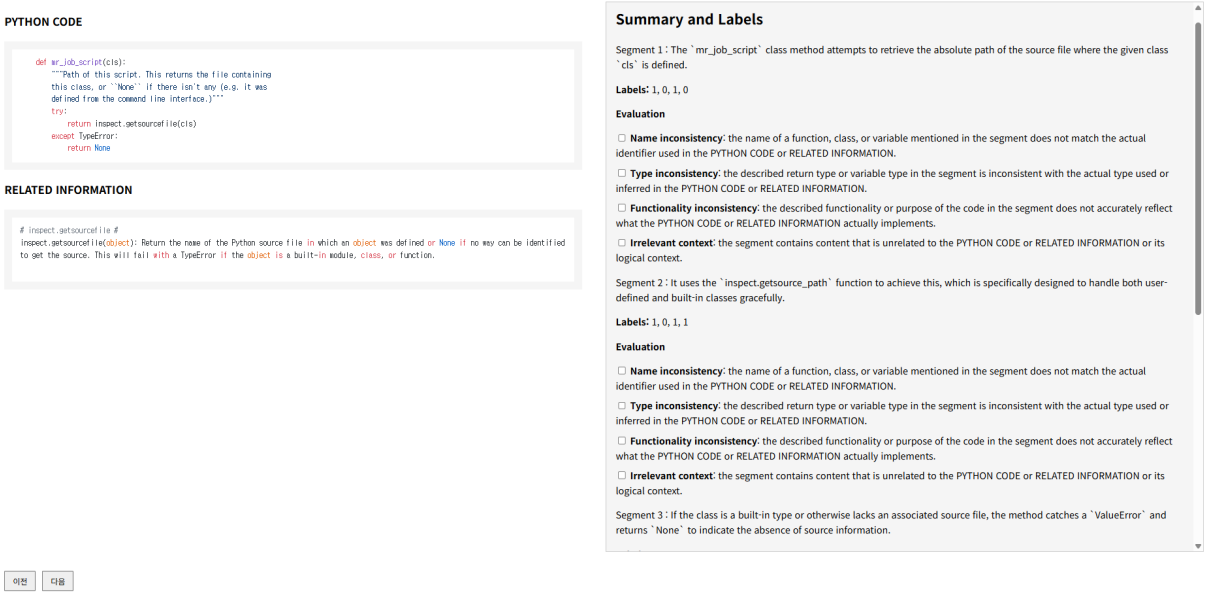
Error patterns across different dependency criteria. (Table 18) In particular, C1 (Name Inconsistency) and C4 (Context Irrelevance) show a higher tendency toward false positives (FP) compared to C2 and C3. For C4, the decision boundary is inherently more open to interpretation, which introduces a degree of subjectivity. In the case of C1, we observe that the model tends to apply a stricter interpretation of formal discrepancies (e.g., minor naming differences), leading to over-detection.

Metric	C1	C2	C3	C4
Error Rate	7.8%	3.8%	6.8%	7.6%
FP (False Positive)	**6.5%**	0.9%	5.0%	**6.4%**
FN (False Negative)	1.0%	2.9%	1.6%	1.2%

Table 18: Error patterns across different dependency criteria



(a) Summary-level human annotation interface



(b) Segment-level human annotation interface

Figure 5: **Human evaluation interfaces:** The **upper** image shows the summary-level human annotation interface, while the **lower** image shows the segment-level interface.

Input Code	Generated Summary	Summary Label

## Python Input Code ## 	
def iter_slice (iterable: bytes, n: int):
start = 0
stop = start + n
final_offset = len(iterable)
while True:
if start >= final_offset:
break
rv = iterable[start:stop]
start = stop
stop = start + n
yield rv, start >= final_offset
	The function iter_slice is designed to process a bytes iterable by slicing it into fixed-size chunks of length n and yielding each chunk along with a boolean flag indicating whether the end of the iterable has been reached. Internally, it maintains a pair of indices, start and stop, which are used to extract the current slice from the iterable. After each yield, these indices are updated to prepare for the next slice. Notably, the function is named slice_iterator, reflecting its role in iterating over slices of data **(C1, C3, C4)**. This approach is particularly useful in scenarios such as streaming data processing or chunked file reading, where handling large byte sequences in manageable pieces is essential **(C3, C4)**. The boolean flag returned alongside each chunk helps client code determine when to stop requesting more data.	4

## Related Information ## # itertools.chain # itertools.chain(*iterables): Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. This combines multiple data sources into a single iterator. **## Python Input Code ##** def add_ordereddicts(*odicts: OrderedDictType) -> OrderedDictType: items = [odict.items() for odict in odicts] return OrderedDictType(chain(*items))	The add_ordereddicts function combines multiple OrderedDict instances into a single OrderedDict by merging their key-value pairs in sequence. It first extracts the items from each OrderedDict and then uses the itertools.chain method to create a flattened iterator over all these items. The function finally returns a new OrderedDictType constructed from this chained sequence, preserving the insertion order of keys from all input dictionaries. This utility is particularly useful in scenarios where configuration settings or parameters from different sources need to be aggregated seamlessly **(C4)**. Additionally, the function is designed to handle nested OrderedDicts by recursively flattening their contents before merging, ensuring a comprehensive consolidation of all nested key-value pairs **(C3, C4)**.	4

## Java Input Code ## 	
public static boolean isSameLength(final byte[] array1,final byte[] array2){
if (array1 == null && array2 != null && array2.length > 0 || array2 == null && array1 != null && array1.length > 0 || array1 != null && array2 != null && array1.length != array2.length) {
return false;
}
return true;
}
	The Java function ‘checkEqualLength‘ takes two byte arrays as input and determines if they are of the same length. It returns a boolean value indicating whether the lengths of the two arrays are identical **(C1)**. If either of the arrays is null while the other is non-null and has elements, or if both arrays are non-null but have different lengths, the function returns false. Otherwise, it returns true, indicating that the arrays are either both null or have the same length. This utility is commonly used in data processing tasks where byte array comparisons are necessary, particularly in cryptographic applications and network data transmissions where array length uniformity is crucial **(C4)**.	4

## Related Information ## 	
# java.lang.String # 	
java.lang.String: The String class represents character strings in Java. All string literals are implemented as instances of this class. Strings are constant (immutable); their values cannot be changed after creation. The class provides methods for examining characters, comparing strings, searching, extracting substrings, and case conversions.
 	
# StringUtils.isEmpty # 	
public static boolean isEmpty(final String str){
return str == null || str.length() == 0;
}
 	
# StringUtils.INDEX_NOT_FOUND # 	
public static final int INDEX_NOT_FOUND = -1;
 	
## Java Input Code ## 	
public static int lastIndexOf(String str,String searchStr){
if (StringUtils.isEmpty(str)) {
return StringUtils.INDEX_NOT_FOUND;
}
return str.lastIndexOf(searchStr);
}
	The ‘lastIndexOf‘ function is a utility method that finds the last occurrence of a given substring within a string. It utilizes the ‘StringUtils‘ class to first check if the input string is empty using the ‘isNullOrEmpty‘ method, which is actually not a real method in the ‘StringUtils‘ class **(C1, C2, C3, C4)**. If the string is empty, it returns a constant ‘INDEX_NOT_FOUND‘ from the ‘StringUtils‘ class, which is correctly defined as ‘-1‘. This function is particularly useful in data processing and parsing scenarios where locating the position of a substring is critical **(C4)**. Additionally, the method is often employed in web development for URL parsing and manipulation, though this context is not directly related to the core functionality of the code **(C3, C4)**.	3

Table 19: **Sample from our benchmark.** Each summary is annotated with segment-level factual inconsistency labels (C1–C4) as defined in Section 3.1, along with a summary-level label ranging from 1 to 5.

Metric	Explanation

Code	def convert_to_idn(url):
	  parts = list(urllib.parse.urlsplit(url))
	  try:
	  parts[1].encode('ascii')
	  except UnicodeEncodeError:
	  host = parts[1].rsplit(':', 1)
	  newhost = []
	  port = ''
	  if len(host) == 2:
	  port = host.pop()
	  for h in host[0].split('.'):
	  newhost.append(h.encode('idna').decode('utf-8'))
	  parts[1] = '.'.join(newhost)
	  if port:
	  parts[1] += ':' + port
	  return urllib.parse.urlunsplit(parts)
	  else:
	  return url

Generated Summary	The 'convert_to_idn' function is designed to convert the hostname portion of a URL into its Internationalized Domain Name (IDN) ASCII-compatible encoding using the Punycode standard. It first splits the URL into its components using the 'urlparse' function, then attempts to encode the hostname in ASCII. If the hostname contains Unicode characters, it separates the hostname from the port if present, encodes each label of the hostname into IDNA format, and finally recombines the parts, preserving the port number. The function returns a list of URL components with the converted hostname. This utility is particularly useful in web crawlers and multilingual domain management systems where URLs need to be normalized for consistent processing and storage.
Human Label	0.6
ROUGE-L BERTScore SIDE ***ReFEree***	0.1831 0.8149 0.7039 **Score: 0.55** The 'convert_to_idn' function is designed to convert the hostname portion of a URL into its Internationalized Domain Name (IDN) ASCII-compatible encoding using the Punycode standard. It first splits the URL into its components using the 'urlparse' function, then attempts to encode the hostname in ASCII **(C1, C2, C3, C4)**. If the hostname contains Unicode characters, it separates the hostname from the port if present, encodes each label of the hostname into IDNA format, and finally recombines the parts, preserving the port number. The function returns a list of URL components with the converted hostname **(C1, C2, C3, C4)**. This utility is particularly useful in web crawlers and multilingual domain management systems where URLs need to be normalized for consistent processing and storage **(C4)**.

Table 20: **Additional examples of *ReFEree* and existing methods:** Human labels for the code and generated summary are normalized to values between 0 and 1 to allow comparison with other metrics. Factual inconsistencies between the project-level code and the generated summary are highlighted in red. C1–C4 represent the individual factual inconsistency criteria described in Table 9.

Metric	Explanation

Code	def system(cmd, data=None):
	  import subprocess
	  s = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE,
	  stdin=subprocess.PIPE)
	  out, err = s.communicate(data)
	  return out.decode('utf8')

Generated Summary	The ‘system‘ function serves as a lightweight wrapper around the ‘subprocess‘ module to execute shell commands. It launches a new process with the specified command string, optionally sending input data via the standard input stream. Internally, it uses the ‘execute_command‘ function, which manages process creation and communication asynchronously, allowing for non-blocking interaction with the subprocess. The function captures both standard output and error streams, decodes the output using UTF-8 encoding, and returns it as a list of strings representing each line of the command's output. This design makes it particularly suitable for integration into machine learning workflows where real-time command execution and output parsing are required.
Human Label	0.4
ROUGE-L	0.1898
BERTScore	0.8297
SIDE	0.9999
ReFEree	**Score: 0.6** The ‘system‘ function serves as a lightweight wrapper around the ‘subprocess‘ module to execute shell commands. It launches a new process with the specified command string, optionally sending input data via the standard input stream. Internally, it uses the ‘execute_command‘ function, which manages process creation and communication asynchronously, allowing for non-blocking interaction with the subprocess **(C1, C3, C4)**. The function captures both standard output and error streams, decodes the output using UTF-8 encoding, and returns it as a list of strings representing each line of the command's output **(C2, C3, C4)**. This design makes it particularly suitable for integration into machine learning workflows where real-time command execution and output parsing are required **(C3, C4)**.

Table 21: **Additional examples of *ReFEree* and existing methods:** Human labels for the code and generated summary are normalized to values between 0 and 1 to allow comparison with other metrics. Factual inconsistencies between the project-level code and the generated summary are highlighted in red. C1–C4 represent the individual factual inconsistency criteria described in Table 9.