Rethinking-based Code Summarization with Chain of Comments

Liwen Cao[1,2], **Hongkui He**[1,2], **Hailin Huang**[1,2], **Jiexin Wang**[1,2], **Yi Cai**[1,2,*]
[1]School of Software Engineering, South China University of Technology
[2]Key Laboratory of Big Data and Intelligent Robot
(South China University of Technology) Ministry of Education
***Correspondence:** ycai@scut.edu.cn

Abstract

Automatic code summarization aims to generate concise natural language descriptions (summary) for source code, which can free software developers from the heavy burden of manual commenting and software maintenance. Existing methods focus on learning a direct mapping from pure code to summaries, overlooking the significant heterogeneity gap between code and summary. Moreover, existing methods lack a human-like re-check process to evaluate whether the generated summaries match well with the code. To address these two limitations, we introduce RBCoSum, a novel framework that incorporates the generated Chain Of Comments (COC) as auxiliary intermediate information for the model to bridge the gap between code and summaries. Also, we propose a rethinking process where a learned ranker trained on our constructed ranking dataset scores the extent of matching between the generated summary and the code, selecting the highest-scoring summary to achieve a re-check process. We conduct extensive experiments to evaluate our approach and compare it with other automatic code summarization models as well as multiple code Large Language Models (LLMs). The experimental results show that RBCoSum is effective and outperforms baselines by a large margin. The human evaluation also proves the summaries generated with RBCoSum are more natural, informative, useful, and truthful.

1 Introduction

With the increasing size and complexity of software, billions of lines of source code reside in online repositories. Code summaries provide a clear natural language description for a piece of the source code and allow a programmer to understand the code's purpose without requiring the programmer to read the code itself. Thus, these summaries provide a particularly useful form of documentation for software development.

Unfortunately, code summaries are often mismatched, missing, or outdated in software projects. Additionally, such summaries are particularly expensive to manually author. Therefore, the automatic code summarization task, which automatically generates brief natural language descriptions for code snippets, offers the ability to automate the summarization of codes and free software developers from the tremendous workload of writing comprehensible summaries.

Early works on automatic code summarization typically treat code snippets as a sequence of code tokens and model it with Seq2Seq architecture(Bahdanau et al., 2015; Iyer et al., 2016; Alon et al., 2018). These works only rely on sequential information in code, which ignores the rich syntax (e.g., abstract syntax tree (AST)) of source code. Therefore, many works (Hu et al., 2018a; Zhang et al., 2019) try to address this challenge by encoding code structures with tree-aware networks (Shido et al., 2019) or graph neural networks (GNNs) (LeClair et al., 2020; Zhou et al., 2022), and obtain better performance than the sequence-based methods. Despite achieving notable advancements, they struggle to learn sophisticated representations due to the lack of comprehensive understanding of code (Feng et al., 2020). Consequently, LLMs pre-trained on extensive code data gain a comprehensive knowledge of code, thus opening up a new opportunity to tackle these limitations (Liu et al., 2023; Wang et al., 2021b; Fried et al., 2022; Roziere et al., 2023).

However, there still exists a significant gap between code snippets and summaries that current methods do not handle effectively. Specifically, as highlighted by (Liu et al., 2020), the code snippet and the summary are heterogeneous, which means they have substantially different lexical tokens, semantic, or language structures. This inherent heterogeneity presents a major challenge when attempting to generate the summary directly from code. Additionally, existing methods lack an explicit human-like re-check process, which means that it is unclear whether the generated summary is functionally consistent with the code. Careful human developers don't stop at just writing a summary. They often re-check the written summary to ensure that the summary accurately captures the essence of the code. This motivates our exploration of whether a rethinking process can improve the automatic code summarization models.

In this paper, we propose RBCoSum (**R**ethinking-**B**ased **Co**de **Sum**marization with Chain of Comments), a general framework to address these two limitations. First, to bridge the gap between code and summary, we introduce a **C**hain **O**f **C**omments (COC) as an auxiliary knowledge for the summarizer. The COC consists of several brief descriptions, each describing a small functionality of a block of the code snippet, serving as intermediate information between code and summaries. Figure 1 illustrates an example of COC in the lower-left corner. Second, we propose a rethinking process that re-checks summaries like human developers. To achieve this process, we leverage the code summarization strengths of code LLMs, coupled with a strategic selection of positive and negative samples to create a ranking dataset. Then, we train a ranker on this ranking dataset that outputs a score reflecting the consistency between a code snippet and its generated summary. The highest-scoring summary is then selected as the predicted summary.

To evaluate the effectiveness of the RBCoSum, we conduct extensive experiments on the public dataset XLCoST (Zhu et al., 2022). The results demonstrate that RBCoSum outperforms baselines with respect to three widely-used metrics (BLEU, ROUGE-L, METEOR) and the metric SIM we introduced. We also perform an extensive ablation study on the COC and rethinking to investigate the contribution of different factors. Additionally, we investigate diverse selection strategies of positive and negative summaries for building the ranking data. Finally, we perform a human evaluation and the results further confirm the effectiveness of our approach.

2 Related Work

Code Summarization: Early studies consider code as plain text and use the Seq2Seq networks with attention(Iyer et al., 2016; Hu et al., 2018b; Wei et al., 2019). Since plain text ignores the structural information in code, some works explore the AST of code and attempt the tree-based and graph-based structures for code summarization(Wan et al., 2018; Shi et al., 2021; Choi et al., 2021; Wang et al., 2021a; Wu et al., 2021). More recently, with the widely used pre-training paradigm in NLP, more source code pre-training models show notable performance on code summarization. Typically, code LLMs can be categorized into three architectures: encoder-only models (Feng et al., 2020; Guo et al., 2020), decoder-only models (Nijkamp et al., 2022; Fried et al., 2022; Li et al., 2023; Roziere et al., 2023), and encoder-decoder models (Ahmad et al., 2021; Wang et al., 2021b, 2023). In contrast to prior approaches that concentrate on learning a mapping directly from code to summaries while neglecting the potentially substantial heterogeneity differences between code and summary, we propose leveraging COC to reduce the gap between code and summary in code summarization.

Reranking: Previous works have shown the effectiveness of learned rankers for sample filtering in many domains. (Yin and Neubig, 2019) propose a simple reranker powered by reconstruction and matching features for neural semantic parsing. (Shen et al., 2021) introduce a model to distinguish between correct and incorrect expressions for Math Word Problems. (Cobbe et al., 2021) proposed to train a verifier to rank solutions sampled from fine-tuned language models for Math QA. (Zhang et al., 2023) reranking uses prompting to obtain a Reviewer model, which checks the generated codes against the language instruction. (Shi et al., 2022) select output programs from a generated set of summaries by marginalizing over program implementations that share the same semantics. (Inala et al., 2022) propose a fault-aware neural ranker that can predict the correctness of a sampled code without executing it. It is trained to predict different kinds of execution information such as the compile/runtime error type. Unlike previous works, we focus on automatic code summarization and propose leveraging code LLMs to construct positive and negative summaries for training the ranker.

3 RBCoSum

Figure 1 shows the overview of the RBCoSum framework. It comprises two phases: SCOC (Summarization with Chain Of Comments) and Rethinking. The SCOC phase aims to mitigate the gap

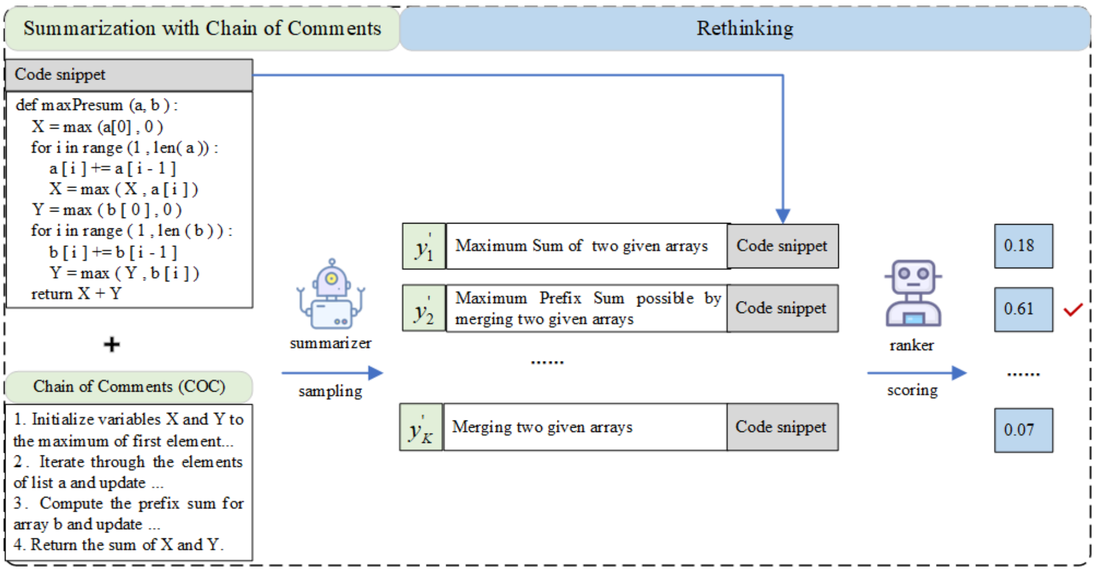

Summarization with Chain of Comments
Rethinking
Code snippet
def maxPresum (a, b) :
X = max (a[0] , 0)
for i in range (1 , len(a)) :
a [i] += a [i - 1]
X = max (X , a [i])
Y = max (b [0] , 0)
for i in range (1 , len (b)) :
b [i] += b [i - 1]
Y = max (Y , b [i])
return X + Y
+
Chain of Comments (COC)
1. Initialize variables X and Y to the maximum of first element...
2. Iterate through the elements of list a and update ...
3. Compute the prefix sum for array b and update ...
4. Return the sum of X and Y.
summarizer
sampling
y_1'
Maximum Sum of two given arrays
Code snippet
y_2'
Maximum Prefix Sum possible by merging two given arrays
Code snippet
......
y_K'
Merging two given arrays
Code snippet
ranker
scoring
0.18
0.61
......
0.07

Figure 1: The illustration of our proposed framework RBCoSum.

between code and summaries by integrating the Chain Of Comments (COC). The role of the re-thinking phase is to act as a re-checker, selecting the summary that best matches the code. In this phase, we first sample the summarizer to generate multiple candidate summaries. Then, a well-trained ranker is adopted to score each candidate summary. The candidate summary with the highest score is selected as the model's predicted summary.

3.1 SCOC

We first introduce how we obtain the COC for code snippets, then discuss the integration of code snippets and COC for fine-tuning the summarizer.

3.1.1 COC generation

Recent advancements in LLMs have demonstrated promising capabilities in understanding intent and following instructions, leading to many investigations utilizing these models as a powerful tool to construct and generate data (Yu et al., 2023; Gunasekar et al., 2023). In light of this, we employ the GPT-3.5-Turbo as a tool to generate COC. The prompt for obtaining the COC is as follows:

Instruction:*"Your goal is to act as a comment generator to produce accurate and clear comment text for each code block within a given code snippet. The requirement is as follows:*
1. Provide comments in list format.
2. Ensure the generated comments effectively capture the essence of the code block.
Code Snippet:
<fill the code snippet here>

3.1.2 Summarization with COC

Based on the constructed COC, we can obtain a new COC-incorporated code summarization dataset D_{cs}, with each instance represented as a code-COC-summary triplet (x, c, y). We then assemble the code snippet and COC into a $prompt_{cs}$ for fine-tuning the summarizer. The input for the summarizer is as follows: "Code Snippet:\n + cs_str + \n + Chain of Comments: + coc_str + \n + So the summary for the code snippet is: + \n", where cs_str and coc_str represent the raw text of code snippet and COC, respectively. We employ the CodeT5-large (Wang et al., 2021b) and Starcoderbase (Li et al., 2023) model as the backbone of the summarizer and fine-tune on the D_{cs} with minimized negative log-likelihood as the loss function:

$$L_{cs}(\theta) = -E_{(x,c,y)\in D_{cs}}[\log p(y|x, c, prompt_{cs}; \theta] \tag{1}$$

3.2 Rethinking

3.2.1 Inference

By sampling from the summarizer, we could obtain multiple candidate summaries. To determine which is the best summary for the code snippet, we propose a rethinking process to learn a scoring model (ranker) $r(x, y')$ that measures how likely the candidate summary y' is the most suitable for the code snippet x. Specifically, during inference, the ranker $r(x, y')$ outputs the highest-score summary y'_{rank} among the set of candidate summaries S:

$$y'_{rank} = \arg\max_{y' \in S} r(x, y'), \quad (2)$$

3.2.2 Ranking Data Creation

First, we collect existing code summarization datasets D, which comprises multiple subsets $\{D_1, D_i, ..., D_I\}$. We also obtain multiple code LLMs M, including $\{M_1, M_j, ..., M_J\}$. Each instance in a subset D_i is represented as a pair (x, y), where x is the code snippet, and y is the reference summary. For each code LLM, we use prompts that are in line with its prompt template in pretraining for better performance. Appendix A shows a prompt example for each model. Then, We sample repetitively from the code LLMs with prompt and temperature T to obtain a summary set $y'_{1:N}$ for the code snippet x. Considering the potential for sampling the same summary, we do a deduplication within $y'_{1:N}$, resulting in a deduplicated summary set $S = y'_{1:K}$. Subsequently, we compute the score of BLEU, ROUGE-L, METEOR, and SIM[1] metrics for each summary and use the mean score value of these four metrics as the ranking metric to represent the quality of summaries and rank them from high to low. We consider the top $\lambda\%$ of summaries as positive samples with the ranking label of v=1, and the others as negative samples with the ranking label of v=0. This way, for each code snippet x, we create a set of training examples $\{(x, y'_k, v_k) | y'_k \in S\}$. Note that we also append $(x, y, v = 1)$ as an additional training example. Following the above process on all instances, we obtain a ranking dataset D_{rank} with 1,390,007 instances[2]. For ease of comprehension, Appendix A outlines the detailed construction process of the ranking dataset and shows the distribution of D_{rank}.

3.2.3 Ranker

We add a classification head on the [CLS] special token of CodeBERT (Feng et al., 2020) to serve as our ranker and further fine-tune the ranker with D_{rank}. Following CodeBERT that inputs a concatenation of two segments with a special separator token [SEP], we use the input of [CLS]; <summary_str >; [SEP]; <code_str>; [EOS]. One segment <summary_str> represents the text of the summary, and <code_str> is the text of the code snippet. The last layer representation of [CLS] token is taken as input to the classification head for predicting the score:

$$d_{cls} = CodeBERT(x; y'), \quad (3)$$

$$r(v|x, y') = d_{cls}W_1 + b_1, \quad (4)$$

3.2.4 Learning Objective

Given the training sample $\{(x, y'_k, v_k) | y'_k \in S\}$ for each input x in D_{rank}, we formulate the loss for input x with the cross-entropy loss between classifier output and labels, normalized by the number of summaries:

$$L_\theta(x, S) = -\frac{1}{|S|} \sum_{y'_k \in S} \log r(v_k | x, y'_k) \quad (5)$$

4 Experimental Setup

4.1 Datasets and Metrics

XLCoST (Zhu et al., 2022) is a dataset containing fine-grained parallel data in commonly used programming languages and summaries. We conduct experiments on two subsets of XLCoST datasets: XLCoST-Java and XLCoST-Python. We follow the divided datasets in previous work (Zhu et al., 2022) where a proportion of training, validation, and testing sets are well-defined. The statistics of these two datasets are listed in Appendix B. To evaluate the quality of generated summaries, we adopt three widely-used metrics, namely BLEU(Papineni et al., 2002), ROUGE-L(Lin, 2004), and METEOR(Lavie and Agarwal, 2007). Furthermore, considering these metrics mainly check the grammar consistency of token sequences, without providing a semantic evaluation, we also introduce another metric-SIM score, in which we utilize sentence-transformers (Reimers and Gurevych, 2019) to encode both the predicted summary and reference summary into distributed representations and compute the cosine similarity between them to obtain the SIM score.

4.2 Implementation Details

To fine-tune the summarizer, we truncate the concatenation of code and COC up to 512 tokens. We adopt AdamW (Loshchilov and Hutter, 2018) with a learning rate of 2e-5 and weight decay of 0.01 to update the model parameters for 10 epochs. The

[1]please refer to section 4.1 for more detail of SIM metric.
[2]https://github.com/galbya/RBCoSum

Methods	XLCoST-Java					XLCoST-Python				
	BLEU	ROUGE-L	METEOR	SIM	AVG	BLEU	ROUGE-L	METEOR	SIM	AVG
Seq2Seq	12.85	20.28	16.15	38.33	21.90	13.62	20.54	17.98	42.01	23.54
Vanilla Transformer	13.70	23.59	20.31	42.38	25.00	14.07	23.44	20.35	43.07	25.23
NeuralCodeSum	12.95	18.27	14.79	38.40	21.10	13.68	22.15	18.70	43.58	24.53
GypSum	16.28	28.17	25.60	52.95	**30.75**	18.17	31.02	28.61	56.52	**33.58**
*Incoder	6.55	2.29	2.09	9.40	5.08	1.68	23.56	20.24	49.91	23.84
*StarCoder	10.58	8.80	8.33	15.32	10.76	2.11	18.13	16.26	39.05	18.89
*Code Llama-Base	10.23	16.07	19.20	37.76	20.81	18.10	21.69	21.93	46.10	26.96
*Code Llama-Instruct	5.54	13.64	22.05	43.47	21.17	4.82	15.44	22.69	52.17	23.78
*Deepseek-Coder-Base	19.02	34.04	36.28	58.66	**37.00**	13.53	23.22	22.04	47.35	26.53
*Deepseek-Coder-Instruct	10.27	23.61	31.66	53.56	29.78	10.11	23.78	31.31	56.27	30.37
*GPT-3.5-Turbo	12.58	25.50	30.28	59.16	31.88	13.03	25.80	29.88	59.04	**31.94**
CodeT5	20.77	36.68	34.16	64.98	39.15	20.16	36.17	32.46	63.41	38.05
CodeT5+COC	21.92	38.60	34.62	65.27	40.10	20.89	36.74	33.65	65.31	39.15
CodeT5+rethinking	20.85	37.22	34.70	66.40	39.80	20.39	36.46	33.93	65.67	39.11
CodeT5+COC+rethinking	22.01	39.14	36.37	67.37	**41.22**††	21.52	37.91	35.66	67.45	**40.64**†
Starcoderbase	20.30	35.87	32.96	63.02	38.04	19.49	35.69	31.62	63.24	37.52
Starcoderbase+COC	21.18	37.54	34.91	65.02	39.66	20.18	35.78	32.24	62.77	37.74
Starcoderbase+rethinking	20.99	37.57	35.51	66.46	40.13	19.96	36.43	33.29	65.13	38.70
Starcoderbase+COC+rethinking	21.55	38.47	36.76	67.42	41.05†	20.16	36.55	34.52	65.66	38.97

Table 1: The comparative results on two datasets. AVG denotes the average score of four metrics. † denote 0.001 $< p <$ 0.01 and †† denote 0.0001 $< p <$ 0.001 in statistical significance testing.

batch size is set to 8, and 16 for Starcoderbase, and CodeT5-large, respectively. For the ranking dataset creation, we adopt nucleus sampling (Holtzman et al., 2019) to sample N=30 summaries for each code snippet from all code LLMs with a temperature T of 0.5, top-p value of 0.95, and max generated tokens of 100. We set λ=20 to obtain positive and negative summaries. For rethinking inference, we sample $|S|$=30 candidate summaries with T=0.5 from the summarizer. We fine-tune the ranker for 20 epochs with a learning rate of 1e-5. To alleviate instability in the model training and temperature sampling, we conduct all experiments three times and report the average results.

4.3 Baselines

We perform a performance comparison across a range from traditional methods to the recent code LLMs: Seq2Seq(Bahdanau et al., 2015), Vanilla Transformer(Vaswani et al., 2017), NeuralCodeSum(Ahmad et al., 2020), GypSum(Wang et al., 2022), CodeT5(Wang et al., 2021b), Starcoderbase(Li et al., 2023), *Incoder(Fried et al., 2022), *StarCoder(Li et al., 2023), *Code Llama-Base(Roziere et al., 2023), *Deepseek-Coder-Base(Guo et al., 2024), *Deepseek-Coder-Instruct(Guo et al., 2024), *GPT-3.5-Turbo(Ouyang et al., 2022). Notably, for the baselines marked with a started *, due to the limitation of computation resources, we employ the zero-shot setting(Brown et al., 2020) to generate summaries without training. Please refer to Appendix C for the details on all baselines.

5 Results Discussion

5.1 Overall Comparison

The overall results are presented in Table 1. CodeT5 performs better compared to Starcoderbase across all evaluation metrics. This may be attributed to the inclusion of a dual-modal generation task during CodeT5's pre-training phase which facilitates the connection between pre-training and code summarization. We also observe that code LLMs with the zero-shot setting generally underperform fine-tuned code LLMs like CodeT5 possibly due to the lack of dataset knowledge. However, some models exhibit remarkable zero-shot code summarization capabilities (Deepseek-Coder-Base vs GypSum). GPT-3.5-Turbo achieves the best performance on XLCoST-Python and demonstrates a more balanced performance across two datasets.

With the incorporation of COC, there is a comprehensive performance boost across all datasets (CodeT5+COC vs CodeT5). The results prove the effectiveness of COC. In Appendix D, we further show that COC enhances the code LLMs without fine-tuning in code summarization. We also compare base code LLMs with models that are equipped with rethinking (CodeT5+rethinking, Starcoderbase+rethinking). We find that the rethinking process improves all the metrics for all base code LLMs despite their different sizes. This indicates that the rethinking process can accurately select the summary that is consistent with the

Methods	XLCoST-Java					XLCoST-Python				
	BLEU	ROUGE-L	METEOR	SIM	AVG	BLEU	ROUGE-L	METEOR	SIM	AVG
CodeT5	20.77	36.68	34.16	64.98	39.15	20.16	36.17	32.46	63.41	38.05
CodeT5+COC	21.92	38.60	34.62	65.27	40.10	20.89	36.74	33.65	65.31	39.15
w/ 25% COC	21.73	38.35	33.99	64.65	39.68	20.52	36.35	33.35	64.44	38.66
w/ 50% COC	21.93	38.43	34.37	64.93	39.92	20.77	36.59	33.33	64.66	38.83
w/ 75% COC	21.85	38.81	34.59	64.99	40.06	21.03	36.80	33.77	65.02	39.15
w/ 25% shuffle	22.01	38.62	34.53	65.46	40.15	20.82	36.77	33.86	65.13	39.14
w/ 50% shuffle	22.13	38.57	34.74	65.36	40.20	20.71	36.40	33.59	65.04	38.93
w/ 75% shuffle	22.22	38.74	34.60	65.36	40.23	20.61	36.54	33.51	64.98	38.91
w/ 100% shuffle	21.94	38.56	34.48	65.34	40.08	20.91	36.64	33.72	64.93	39.05
w/ 25% replace	21.41	37.82	33.86	63.73	39.21	20.42	36.04	33.09	63.70	38.31
w/ 50% replace	21.18	37.20	33.43	62.68	38.62	19.67	35.31	32.13	62.25	37.34
w/ 75% replace	20.87	36.38	32.64	61.33	37.80	19.48	34.74	31.41	61.44	36.77
w/ 100% replace	19.99	34.99	31.52	58.79	36.32	18.98	33.78	30.39	59.36	35.63

Table 2: Contribution Investigation of COC.

code's functionality. Another interesting observation is that the performance gains on the SIM metric are more notable. This is probably because the SIM metric carries more weight in the ranking metric comparing other metrics.[3]

We build the CodeT5+COC+rethinking and Starcoderbase+COC+rethinking to understand the effect when both COC and rethinking are integrated, which serve as the best version of our approach, see the last two lines in Table 1. As expected, we can see that the combination of COC and rethinking further improves the performance on all metrics for both datasets. This suggests the ability of the summarizer to benefit not only from COC but also from rethinking.

5.2 Contribution Investigation of COC

The specific aspects of the COC that lead to improved performance remain unclear. Therefore, we perform experiments on the COC from three aspects: quantity, order, and quality to understand their impact. To investigate the impact of COC quantity, we feed the summarizer with only a specific proportion of sentences in COC. For instance, w/ 25% COC denotes that we only use the initial 25% of sentences from the full COC. For the order aspect, we randomly select a certain percentage of sentences from the COC and shuffle these sentences while keeping the rest unchanged. For example, "w/ 25% shuffle" represents the random selection of 25% of sentences in COC, and the order of these sentences is shuffled, while the remaining sentences maintain their original order. We control the quality of COC by changing the semantics of some sentences in the full COC. In particular, we select a certain proportion of sentences and replace them with an equivalent number of sentences from the test set COC pool (randomly chosen, sentences may come from the same COC or different COCs). In Appendix E, we further discuss the direct impact of COC quality on the performance of code summarization.

Table 2 shows several interesting results. First, we find that as the number of retained sentences in the COC increases, the AVG performance of the model slightly improves (w/ 50% COC vs. w/ 25% COC). This suggests that with more detailed descriptions from the COC, the model can better understand the code's functionality, further validating the effectiveness of introducing COC. Moreover, performance gains are minimal when the quantity is up 75%. This indicates that a COC containing 75% of the information may already cover all the important details of the code.

Additionally, an interesting finding is that in the XLCoST-Java dataset, the performance of w/25% random, w/50% random, and w75% random is very similar. This indicates that the summarizer is not sensitive to the sentence order in the COC, as long as the overall semantics remain consistent. The model retains its ability to extract useful information even from the disordered COC. Another finding is that the performance significantly declines as the replacement ratio of sentences in the COC increases. This demonstrates the importance of introducing high-quality COC. When the replacement ratio exceeds 50% in both datasets, the AVG performance becomes worse than that of CodeT5. This suggests that the model can understand and learn useful information from the COC. But, at the

[3]We hypothesize assigning different weights to different metrics may be more appropriate. We call for future work.

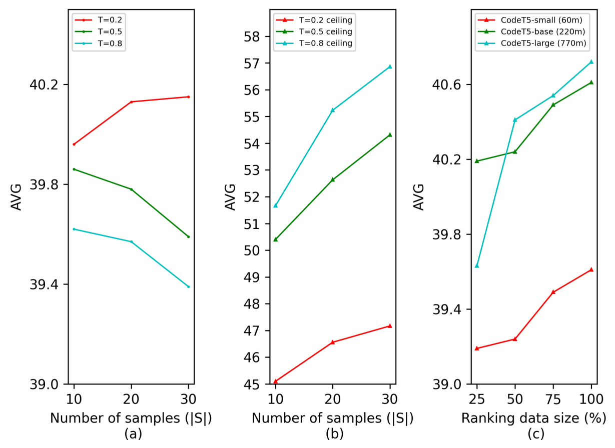


Figure 2: (a) AVG performance as a function of the number of samples ($|S|$) for various temperature settings. (b) Ceiling AVG performance against the $|S|$ with various temperatures. (c) Comparison of AVG performance of ranker with varying model sizes on various ranking data sizes.

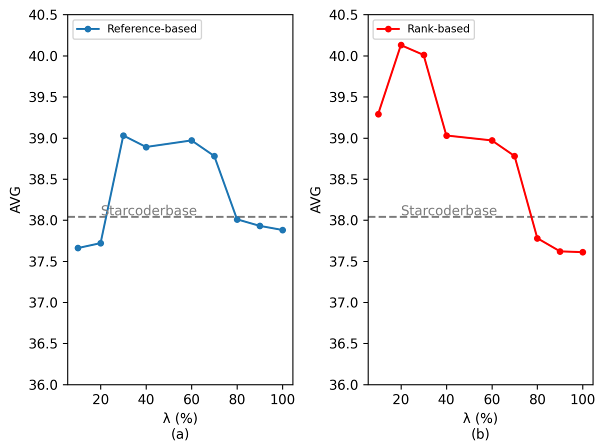


Figure 3: AVG performance against λ for StarCoder-base+rethinking on XLCoST-Java dataset.

same time, wrong COCs can mislead the model into generating incorrect summaries.

5.3 Ablation Study of Rethinking

To investigate the impact of rethinking, we evaluate it based on sampling temperature (T), number of samples ($|S|$), ranking dataset size, and ranker model size. Figure 2 (a) shows AVG performance against $|S|$ and T. At T=0.2, AVG performance improves with more samples, but at T=0.5 and T=0.8, performance declines as sampling size increases. This suggests that low temperatures yield deterministic results with limited performance gains from more samples, while higher temperatures introduce diversity but also more negative samples, which can confuse the ranker and reduce performance.

Furthermore, to explore the ceiling performance of the ranker, we compute the maximum AVG score within the candidate summary set and average it across the entire test set to establish a ceiling performance (the performance ceiling when the ranker could rank the best candidate summary for all code snippets and we can access to reference summaries). Figure 2 (b) presents findings that there are significant performance gains at high temperatures and large sample sizes for ceiling performance, with higher temperatures yielding larger improvements. This suggests our trained ranker model lacks robustness at high temperatures. Enhancing its performance at high temperatures could mitigate the performance degradation seen in Figure 2 (a) at high temperatures. This calls for future improvements.

To explore the impact of ranking dataset size and model size of ranker, we randomly sampled different proportions (from 25% to 100% with 25% increments) samples from the ranking data D_{rank} to train rankers with different parameter sizes ranging from 60m to 770m. We can draw the following conclusions from Figure 2: 1) As the size of the ranking dataset increases, the performance shows improvement and seems not yet saturated, indicating that increasing the data size is an effective approach to enhancing the ranker's ability to recognize high-quality summaries. 2) Larger-size rankers benefit more from larger ranking data sizes (CodeT5-large obtains more improvement than CodeT5-small).

5.4 The Study of Strategies for Positive and Negative Sample Selection

We select the top 20% (λ=20) of these summaries as positive summaries and others as negative summaries. In this section, we examine the reason behind this choice and explore the alternative selection strategies. In general, there are two types of selection strategies: 1) Reference-based selection, where only the reference summary is positive, and varying proportions of the last-λ% generated summaries are negative; 2) Rank-based selection, where we select the top λ% generated summaries as positive summaries and the others as negative. As can be seen from Figure 3 (a), for the Reference-based selection, the performance generally increases with the addition of more relevant summaries, but it drops sharply when reaching 80%. This suggests that the top 20% of summaries may be very similar to the reference summary, and it is unreasonable to treat them as negative summaries. Rank-based selection (Figure 3 (b)) shows that using higher-ranked summaries as positives boosts performance, peaking at 20%, but drops sharply after introducing more irrelevant summaries as positive summaries. This suggests that introducing the top 20% summaries as negative samples can hinder the model's ability to identify the quality of summaries.

5.5 Human evaluation

The ultimate goal of the automatic code summarization model is to help developers understand the functionality of the code. Therefore, in this section, we conduct a human evaluation to assess the generated summaries. Following(Li et al., 2021), we first establish four criteria: 1) Naturalness: grammaticality and fluency of the summary. 2) Informativeness: the amount of content carried over from the input code snippets to the summary, ignoring fluency. 3) Usefulness: how useful the summary is for developers in understanding code. 4) Truthfulness: We propose this new criterion for automatic code summarization to measure the hallucination of code LLMs considering the prevalence of hallucination in language models.

We randomly select 100 code snippets from the test set of each dataset, forming a total of 200 code snippets. Subsequently, we employ three models—Starcoderbase, Starcoderbase+COC, and Starcoderbase+COC+rethinking—to generate summaries for each code snippet. We then assemble a team of non-co-authors evaluators, including two Ph.D. students and two master students in software engineering with 2-5 years of Python and Java programming experience. Each evaluator is invited to score the summaries on a scale of 0 to 4 from four established criteria. Note that we show evaluators summaries generated by three models at a time, making it easier and faster for evaluators to compare the gap among different summaries and score more fairly. To mitigate subjective variations among evaluators, the final score for each model is the average of all evaluators' scores. Results are shown in Figure 4.

We find that Starcoderbase+COC consistently outperforms Starcoderbase across all criteria, with notable improvements in Informativeness and Usefulness. This reveals that the introduction of COC primarily enhances the Informativeness and Usefulness and the code summarization model can learn

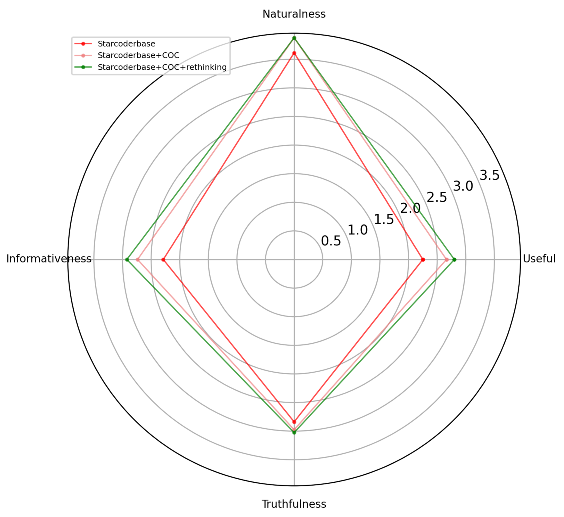
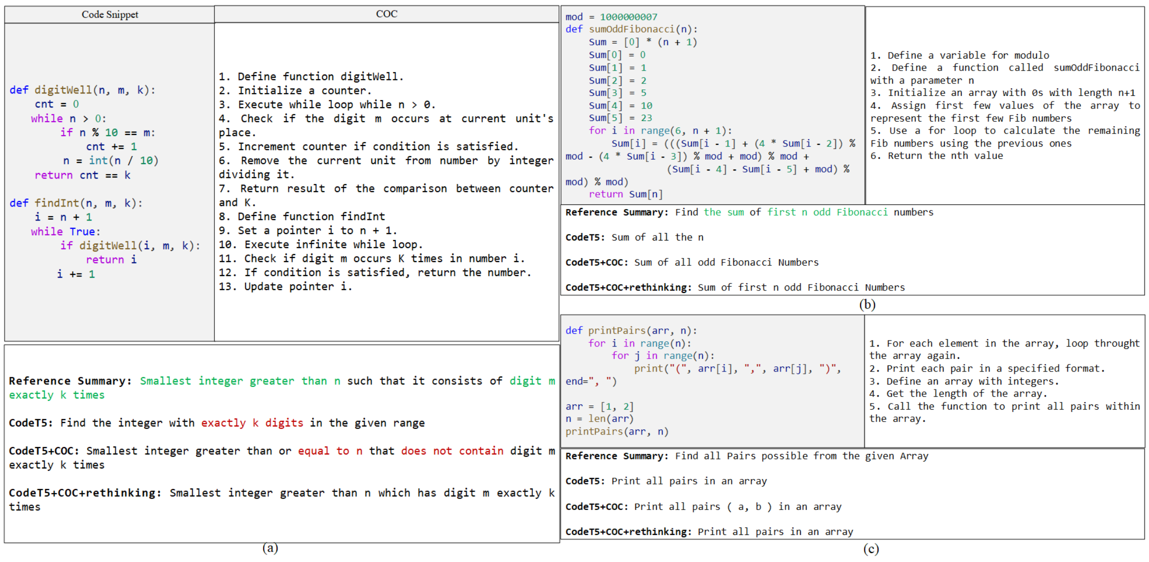


Figure 4: Radar plot for human evaluation.

crucial details from COC to enrich the content of summaries. Additionally, COC slightly improves Truthfulness, likely due to its factual descriptions. Introducing COC into code LLMs may be a promising method for mitigating the hallucination of code LLMs for code summarization. Furthermore, compared to the Starcoderbase+COC, the Starcoderbase+COC+rethinking shows performance gains in Informativeness and Useful, with minimal improvement in Naturalness and Truthfulness. This is likely due to the stronger correlation between our ranking metric and Informativeness and Useful criteria.

6 More Examaples for Case Study

Figure 5(a) and Figure 5(b) show two cases of generated summaries under three settings: CodeT5, CodeT5+COC, and CodeT5+COC+rethinking. In these two cases, CodeT5 struggles to capture the key information in the code, while CodeT5+COC and CodeT5+COC+rethinking successfully identify the essential details. For instance, in Figure 5(a), the reference summary highlights two key parts: "Smallest integer greater than n" and "digit m exactly k times". The summary generated by CodeT5 misunderstands "k times" as "k digits" and fails to mention the important detail "Smallest integer greater than n". On the other hand, CodeT5+COC does include these two key information but both are incorrect. First, the code aims to find a number "greater than n", rather than "equal to n". Second, the goal is to find an integer that "contains" the number m exactly k times, not the one that "does not contain" it. These



Code Snippet | COC

```
def digitWell(n, m, k):
    cnt = 0
    while n > 0:
        if n % 10 == m:
            cnt += 1
        n = int(n / 10)
    return cnt == k

def findInt(n, m, k):
    i = n + 1
    while True:
        if digitWell(i, m, k):
            return i
        i += 1
```

1. Define function digitWell.
2. Initialize a counter.
3. Execute while loop while n > 0.
4. Check if the digit m occurs at current unit's place.
5. Increment counter if condition is satisfied.
6. Remove the current unit from number by integer dividing it.
7. Return result of the comparison between counter and K.
8. Define function findInt
9. Set a pointer i to n + 1.
10. Execute infinite while loop.
11. Check if digit m occurs K times in number i.
12. If condition is satisfied, return the number.
13. Update pointer i.

Reference Summary: Smallest integer greater than n such that it consists of digit m exactly k times

CodeT5: Find the integer with exactly k digits in the given range

CodeT5+COC: Smallest integer greater than or equal to n that does not contain digit m exactly k times

CodeT5+COC+rethinking: Smallest integer greater than n which has digit m exactly k times

(a)

```
mod = 1000000007
def sumOddFibonacci(n):
    Sum = [0] * (n + 1)
    Sum[0] = 0
    Sum[1] = 1
    Sum[2] = 2
    Sum[3] = 5
    Sum[4] = 10
    Sum[5] = 23
    for i in range(6, n + 1):
        Sum[i] = (((Sum[i - 1] + (4 * Sum[i - 2]) % 
mod - (4 * Sum[i - 3]) % mod + mod) % mod + 
                (Sum[i - 4] - Sum[i - 5] + mod) % 
mod) % mod)
    return Sum[n]
```

1. Define a variable for modulo
2. Define a function called sumOddFibonacci with a parameter n
3. Initialize an array with 0s with length n+1
4. Assign first few values of the array to represent the first few Fib numbers
5. Use a for loop to calculate the remaining Fib numbers using the previous ones
6. Return the nth value

Reference Summary: Find the sum of first n odd Fibonacci numbers

CodeT5: Sum of all the n

CodeT5+COC: Sum of all odd Fibonacci Numbers

CodeT5+COC+rethinking: Sum of first n odd Fibonacci Numbers

(b)

```
def printPairs(arr, n):
    for i in range(n):
        for j in range(n):
            print("(", arr[i], ",", arr[j], ")", 
end=", ")

arr = [1, 2]
n = len(arr)
printPairs(arr, n)
```

1. For each element in the array, loop throught the array again.
2. Print each pair in a specified format.
3. Define an array with integers.
4. Get the length of the array.
5. Call the function to print all pairs within the array.

Reference Summary: Find all Pairs possible from the given Array

CodeT5: Print all pairs in an array

CodeT5+COC: Print all pairs (a, b) in an array

CodeT5+COC+rethinking: Print all pairs in an array

(c)

Figure 5: Three cases from the XLCoST-Python benchmark.

cases demonstrate that while COC can enhance the amount of key information in the summary, it may also introduce errors. Compared to CodeT5+COC, CodeT5+COC+rethinking filters out these two erroneous pieces of information, ensuring a more accurate summary. Figure 5(c) presents a case where all three settings generate correct summaries. It can be observed that the code in this case is relatively simple and does not involve complex terms like Fibonacci Numbers in Figure 5(b). In such cases, there may be no need to introduce COC. Therefore, we can only activate COC in more complex code to provide additional informative context, which helps reduce the cost of COC generation.

7 Conclusions And Future Work

In this paper, we introduce the RBCoSum framework for automatic code summarization. It reduces the substantial gap between code and summaries by incorporating the Chain of Comments (COC) into the model. Moreover, we propose a rethinking process that utilizes a learned ranker as an explicit mechanism for re-checking the alignment between the generated summary and the code. Also, we propose to leverage code LLMs to build positive and negative samples for training the ranker. The extensive experiments show that RBCoSum achieves consistent and strong improvement on the XLCoST-Java and XLCoST-Python datasets. In the future, we will explore multiple finer-grained dimensions for the rethinking process rather than solely relying on the digital score.

8 Limitations

Although we validate the effectiveness of RBCoSum in both Java and Python programming languages, including additional programming languages would more clearly demonstrate that the proposed method is language-agnostic.

Acknowledgements

This research is supported by the Science and Technology Planning Project of Guangdong Province (2020B0101100002), the National Natural Science Foundation of China (62076100, 62476097, 62402185), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2240100), Guangdong Provincial Fund for Basic and Applied Basic Research—Regional Joint Fund Project (Key Project) (2023B1515120078), Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), the China Computer Federation (CCF)-Zhipu AI Large Model Fund.

References

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.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings* *of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. In *International Conference on Learning Representations*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred api knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2269–2275.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 2073–2083.

Denis Kocetkov, Raymond Li, LI Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. 2022. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*.

Alon Lavie and Abhaya Agarwal. 2007. Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 228–231.

Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, pages 184–195.

Jia Allen Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 155–166.

Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, LI Jia, Jenny Chim, Qian Liu, et al. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. Refbert: A two-stage pre-trained framework for automatic rename refactoring. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 740–752.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. In *International Conference on Learning Representations*.

Ilya Loshchilov and Frank Hutter. 2018. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Antonio Valerio Miceli-Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

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.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

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.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Jianhao Shen, Yichun Yin, Lin Li, Lifeng Shang, Xin Jiang, Ming Zhang, and Qun Liu. 2021. Generate & rank: A multi-task framework for math word problems. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2269–2279.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4053–4062.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407.

Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2021a. Cocosum: Contextual code summarization with multi-relational graph neural network. *arXiv preprint arXiv:2107.01933*.

Yu Wang, Yu Dong, Xuesong Lu, and Aoying Zhou. 2022. Gypsum: learning hybrid representations for code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 12–23.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven Hoi. 2023.

Codet5+: Open code large language models for code understanding and generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021b. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems*.

Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code summarization with structure-induced transformer. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1078–1090.

Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187*.

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, Tao Yu, Tatsunori B Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida I Wang. 2023. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*, pages 41832–41846.

Yu Zhou, Juanjuan Shen, Xiaoqing Zhang, Wenhua Yang, Tingting Han, and Taolue Chen. 2022. Automatic source code summarization with graph attention networks. *Journal of Systems and Software*, 188:111257.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. In *Deep Learning for Code Workshop*.

A Ranking Data Creation Procedure and Statistics

Algorithm 1 outlines the detailed construction process of the ranking dataset. Table 4 shows the distribution of the ranking dataset obtained by four different code LLMs (StarCoder(Li et al., 2023), Code Llama(Roziere et al., 2023), CodeT5+ (Wang et al., 2023) and Incoder(Fried et al., 2022)) and four code summarization datasets (CodeSearchNet(Husain et al., 2019), XLCoST-Java, XLCoST-Python, and code-docstring-corpus(Miceli-Barone and Sennrich, 2017). For these four models, we use the prompt aligned with the prompt format employed during model pre-training for each model to optimal summary generation. Table 3 shows a prompt example for each model. We will open-

Model	Prompt example
StarCoder	"<fim_prefix>def hello_world():\n""" <fim_suffix>""" \n\tprint("hello world!") <fim_middle>"
Code Llama	"<PRE>def hello_world():\n\t""" <SUF>"""\n\t print("hello world!") <MID>"
CodeT5+	"def hello_world():\n\tprint("hello world!")
Incoder	"def hello_world():\n\t""" <|mask:0|>"""\n\t print("hello world!")<|mask:1|><|mask:0|>

Table 3: Prompt example for each code LLM.

Algorithm 1 The creation procedure of ranking dataset

```
Input: Code summarization datasets $D = \{D_1, D_i, ..., D_I\}$, and code LLMs $M = \{M_1, M_j, ..., M_J\}$
Output: Ranking dataset $D_{rank}$
for each instance $(x, y)$ in $D_i$ do
    for each model $M_j$ in $M$ do
        Obtain a set of $K$ unique candidate summaries $y'_{1:K}$, by first sampling $N$ summaries $y'_{1:N}$ from $M_j$ and then removing all duplicated summaries.
        Compute the ranking metric based on $y$ and each $y'$ and sort $y'_{1:K}$ with the ranking metric from highest to lowest.
        for each candidate summary $y'_k$ in $y'_{1:K}$ do
            Get the binary ranking label for $y'_k$ with the following defined rules ($\lambda \in (0, 1]$ and we perform an upward rounding of the value of $\lambda K$):
                $$v_k = \begin{cases} 1 & \text{if } k \in [1, \lambda K]] \\ 0 & \text{if } k \in [\lambda K, K] \end{cases}$$
            Append the ranking training samples $\{(x, y'_k, v_k)|y'_k \in S\}$ for code snippet $x$ to the ranking dataset $D_{rank}$.
        end for
    end for
end for
```

Dataset	StarCoder		Code Llama		CodeT5+		Incoder		All
	Train	Valid	Train	Valid	Train	Valid	Train	Valid	
CodeSearchNet	157,147	6,332	77,665	4,118	70,465	4,005	80,434	4,275	404,468
XLCoST-Java	195,045	9,317	94,970	4,176	98,323	4,349	85,911	3,798	495,889
XLCoST-Python	187,110	8,614	68,285	3,155	31,655	1,356	35,304	1,595	337,074
Code-docstring-corpus	28,516	1,510	58,787	3,103	27,435	1,454	30,172	1,599	152,576
All	567,845	25,773	299,707	14,552	227,878	11,164	231,821	11,267	**1,390,007**

Table 4: The distribution of created ranking dataset using four code LLMs and four code summarization datasets. We construct ranking data for XLCoST-Java and XLCoST-Python using only the code in the training set.

Dataset	PL	# Training	# Validation	# Test	Lines per code	Tokens per code	Tokens per summary
XLCoST-Java	Java	9,623	494	911	3.71	227.09	10.67
XLCoST-Python	Python	9,263	472	887	3.82	215.29	10.70

Table 5: Statistics of XLCoST-Java and XLCoST-Python datasets.

source our source code and the collected ranking dataset to the community and this large ranking dataset can serve as a valuable pre-training resource.

B Dataset Statistics

The statistics of XLCoST-Java and XLCoST-Python datasets are listed in Table 5

C Baselines

- **Seq2Seq (Bahdanau et al., 2015)** uses an LSTM-based encoder-decoder architecture with an attention mechanism to learn from the code and generate summaries.

- **Vanilla Transformer (Vaswani et al., 2017)** adopts a Transformer encoder-decoder architecture with 6 layers of the encoder and 6 layers of the decoder. We directly tune it from scratch.

- **NeuralCodeSum (Ahmad et al., 2020)** takes the Transformer structure and replaces the original positional encoding with the relative positional encoding, which encodes the pairwise relationships between the tokens in the source code text.

- **GypSum (Wang et al., 2022)** introduces particular edges related to the control flow of a code snippet into the abstract syntax tree for graph construction and designs two encoders to learn from the graph and the token sequence of source code, respectively.

- **CodeT5 (Wang et al., 2021b)** builds on an encoder-decoder Transformer model with the same architecture as T5 (Raffel et al., 2020) and makes better use of the code semantics conveyed from developer-assigned identifiers. We directly tune CodeT5-large with the training set for code summarization

- **Starcoderbase (Li et al., 2023)** is an ensemble of code LLMs with parameter sizes ranging from 1B to 7B. It is trained on 80+ programming languages from The Stack (Kocetkov et al., 2022), utilizing the fill-in-the-middle objective. We use the Starcoderbase model with 1B parameters and directly tune it with the training set.

- ***Incoder (Fried et al., 2022)** is pre-trained on a mixture of multilingual code data from GitHub and StackOverflow posts, utilizing a causal masking objective. This model possesses both code understanding and generation capabilities. We utilize the InCoder model with 6.7B parameters.

- ***StarCoder (Li et al., 2023)** is a 15.5B parameter model with an 8K window size and FIM (Fill In the Middle, or infilling) capability. It outperforms many previous open-source large language models that support code summarization.

- ***Code Llama-Base (Roziere et al., 2023)** is a family of LLMs for code generation and infilling, ranging in scale from 7B to 34B parameters. All Code Llama models are initialized with Llama 2 (Touvron et al., 2023) model weights and trained on 500B tokens from a code-heavy dataset. We use the 7B versions of Code Llama from the Hugging Face.

Models	XLCoST-Java			XLCoST-Python		
	BLEU	ROUGE-L	METEOR	BLEU	ROUGE-L	METEOR
Code Llama-Instruct	5.54	13.64	22.05	4.82	15.44	22.69
Code Llama-Instruct+COC	8.78	18.69	24.31	8.36	19.80	24.79
DeepSeek Coder-Instruct	10.27	23.61	31.66	10.11	23.78	31.31
DeepSeek Coder-Instruct+COC	11.23	25.78	31.76	11.72	26.62	31.83
GPT-3.5-Turbo	12.58	25.50	30.28	13.03	25.80	29.88
GPT-3.5-Turbo + COC	13.11	26.33	31.33	14.42	27.29	31.18

Table 6: Code Summarization performance of LLMs+COC.

Models	COC score	BLEU	ROUGE-L	METEOR
Code Llama-Instruct	2.05	20.1	35.38	32.43
DeepSeek V2-Chat	2.7	21.34	37.22	34.62
GPT-3.5-Turbo	2.85	21.92	38.6	34.75

Table 7: Investigation of summarization performance against the COC quality on XLCoST-Java.

- ***Code Llama-Instruct (Roziere et al., 2023)** The Code Llama-Instruct models are based on Code Llama and fine-tuned with an additional 5B tokens to better follow human instructions. We also use the 7B version of Code Llama-Instruct.

- ***Deepseek-Coder-Base (Guo et al., 2024)** is composed of a series of code LLMs. Each model is pre-trained on 2T tokens from scratch and employs an extra fill-in-the-blank task to support infilling abilities. We use the 6.7B version of Deepseek-Coder-Base.

- ***Deepseek-Coder-Instruct (Guo et al., 2024)** is initialized from Deepseek-Coder-Base and fine-tuned on 2B tokens of instruction data.

- ***GPT-3.5-Turbo (Ouyang et al., 2022)** has been trained on a diverse range of internet text, enabling it to demonstrate impressive understanding and generation capabilities in both language and code.

D LLMs+COC Without Fine-tuning

Section 5.1 demonstrates that COC enhances the summarization performance of code LLMs under fine-tuning settings. In this section, we explore the effects of integrating COC into LLMs under zero-shot prompting. Concretely, we prompt the instruct-based LLMs to consider both the code and the COC generated by GPT-3.5-Turbo with the following input: *"Examine the code snippet and its COC (chain of comments). Provide a brief summary that captures the essence of the code.*
Code Snippet:<*fill the code snippet here*>
COC:<*fill the COC here*>

As shown in Table 6, we find that LLMs+COC improve summarization performance by a large margin. This enhancement not only underscores the effectiveness of combining LLMs with COC but also confirms the robustness of COC in generalizing across different contexts. We also observe the most significant improvement in the Code Llama-Instruct model. This potentially suggests that the COC provides more pronounced benefits to LLMs with weaker baseline performance.

E Impact of COC Quality on Summarization Performance

Due to the lack of labeled COC, it is challenging to directly analyze the impact of COC quality on summarization performance using conventional metrics like BLEU. As a result, in Section 5.2, we manipulate the quality of COC indirectly by incorporating irrelevant COC sentences. Here, we conduct a manual evaluation of COC quality to investigate its direct influence on the summarizer. We prompt Code Llama-Instruct, DeepSeek V2-Chat(Liu et al., 2024), and GPT-3.5-Turbo to generate COC, with CodeT5+COC as the summarizer. Then, we randomly select 100 instances and manually score the generated COC based on accuracy criteria (to what extent the COC accurately captures the essence of code block) rated from 0 to 4. The COC score for is obtained by averaging the accuracy scores across all evaluators. As shown in Table 7, we observe a strong correlation between the quality of the COC and the performance of code summarization. This demonstrates that our proposed COC is effective and that introducing high-quality COC has a significantly positive impact on the performance of code summarization.