Automatic Code Summarization via ChatGPT: How Far Are We?

Weisong Sun[1,2], Chunrong Fang[1]*, Yudu You[1], Yun Miao[1], Yi Liu[2], Yuekang Li[3], Gelei Deng[2],
Shenghan Huang[1], Yuchen Chen[1], Quanjun Zhang[1], Hanwei Qian[1], Yang Liu[2], Zhenyu Chen[1]

[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[2]Nanyang Technological University, Singapore
[3] University of New South Wales, Australia
*fangchunrong@nju.edu.cn

ABSTRACT

To support software developers in understanding and maintaining programs, various automatic code summarization techniques have been proposed to generate a concise natural language comment for a given code snippet. Recently, the emergence of large language models (LLMs) has led to a great boost in the performance of natural language processing tasks. Among them, ChatGPT is the most popular one which has attracted wide attention from the software engineering community. However, it still remains unclear how ChatGPT performs in (automatic) code summarization. Therefore, in this paper, we focus on evaluating ChatGPT on a widely-used Python dataset called CSN-Python and comparing it with several state-of-the-art (SOTA) code summarization models. Specifically, we first explore an appropriate prompt to guide ChatGPT to generate in-distribution comments. Then, we use such a prompt to ask ChatGPT to generate comments for all code snippets in the CSN-Python test set. We adopt three widely-used metrics (including BLEU, METEOR, and ROUGE-L) to measure the quality of the comments generated by ChatGPT and SOTA models (including NCS, CodeBERT, and CodeT5). The experimental results show that in terms of BLEU and ROUGE-L, ChatGPT's code summarization performance is significantly worse than all three SOTA models. We also present some cases and discuss the advantages and disadvantages of ChatGPT in code summarization. Based on the findings, we outline several open challenges and opportunities in ChatGPT-based code summarization.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Large language model, ChatGPT, automatic code summarization, neural code summarization

*Corresponding author.

1 INTRODUCTION

Code comments play a critical role in facilitating program comprehension [54, 63] and software maintenance [11, 23]. It is a good programming practice to write high-quality code comments [11, 53, 67]. However, writing comments is a labor-intensive and time-consuming task. Consequently, high-quality comments are often absent, unmatched, and outdated during software evolution. Lack of high-quality code comments is a common problem in the software industry [27, 28]. Automatic code summarization (code summarization, for short) is a hot research topic [9, 47, 48], which aims to design advanced techniques to support the automatic generation of code summaries (i.e., comments). For a code snippet (e.g., a Java method or a Python function) given by the developer, code summarization techniques can automatically generate natural language summaries for it.

In recent years, a large number of deep learning (DL)-based code summarization techniques have been proposed one after another, such as DeepCom [25], NCS [1], and SIT [64]. They leverage powerful generative models trained on a large-scale code-comment corpus to translate code snippets in programming languages into summaries in natural language [2, 18, 25, 30, 32, 48]. Leveraging powerful deep learning, these techniques have shown to outperform traditional information retrieval (IR)-based techniques [20, 21, 39, 40, 52] in terms of the naturalness and informativeness of the generated summaries [25, 33, 62]. After that, with the success of the pretraining and fine-tuning paradigm in the field of NLP (e.g., BERT and T5), many works in software engineering (SE) introduce this paradigm to further boost code-related tasks, including code summarization (e.g, CodeBERT [15] and CodeT5 [61]). In practice, these works first pre-train a large-scale language model (i.e., LLM) with general language modeling tasks, such as masked language modeling (MLM) and unidirectional language modeling (ULM). Then, they fine-tune the pre-trained models on downstream tasks, such as code clone detection, code search, and code summarization.

Most recently, an intelligent human-machine dialogue LLM called ChatGPT has attracted great attention. ChatGPT is created by finetuning a GPT-3.5 series model via reinforcement learning from human feedback (RLHF) [10]. Compared with previous DL approaches,
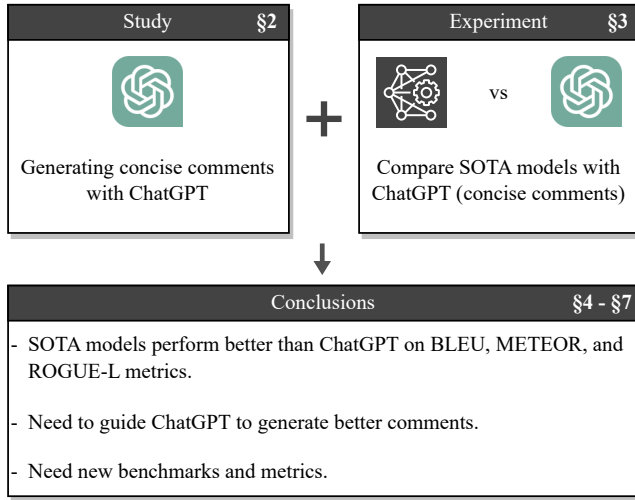
Figure 1: Overview of this paper

ChatGPT accepts longer input and enables users to have a conversation with the language model while the previous chat history is taken into account when generating answers. With the emergence of ChatGPT, there is growing interest in leveraging this model for various NLP tasks [5, 31, 43, 65]. In addition to natural language, ChatGPT can also deal with source code, which arouses growing interest in applying ChatGPT to SE tasks [8, 13, 14, 50]. However, the exploration of ChatGPT in code summarization is still lacking.

In this paper, we perform a preliminary evaluation of ChatGPT's zero-short code summarization performance. The overview of this study is depicted in Figure 1. The overall idea is to explore the potential of ChatGPT in code summarization and compare the explored strategy of using ChatGPT with state-of-the-art (SOTA) techniques to draw conclusions. Specifically, we first design several heuristic questions to collect the feedback of ChatGPT. Based on the feedback, we find an appropriate prompt to guide ChatGPT to generate in-distribution comments. Then, we use such a prompt to ask ChatGPT to generate comments for all 14,918 code snippets in the CSN-Python test set. We leverage three widely-used metrics (i.e., BLEU, METEOR, ROUGE-L) to measure the quality of the comments generated by ChatGPT compared to ground-truth comments. Moreover, to provide a deeper understanding of ChatGPT's code summarization performance, we provide a comparison with results obtained by three SOTA code summarization models (including NCS [1], CodeBERT [15], and CodeT5 [61]). The experimental results show that overall ChatGPT is inferior to the three SOTA code summarization models in terms of BLEU, METEOR, and ROUGE-L. We also show and analyze some successful, moderate, and weak cases of ChatGPT. Based on the findings, we outline several challenges and opportunities in ChatGPT-based code summarization that remain to be addressed in future work, including how to improve ChatGPT on existing benchmark datasets and metrics, how to crop the rich but lengthy comments generated by ChatGPT, how to create a higher-quality benchmark dataset by combing existing datasets with ChatGPT-generated comments, and how to design

suitable metrics to evaluate ChatGPT-based code summarization models.

In summary, we make the following contributions.

- To the best of our knowledge, we are the first to investigate and analyze ChatGPT's code summarization performance.
- We find an appropriate prompt to guide ChatGPT to generate in-distribution comments through carefully crafted heuristic questions and feedback.
- We evaluate ChatGPT on a widely-used code summarization dataset and compare it with three SOTA code summarization models.
- Based on the findings, we outline several challenges and opportunities for ChatGPT-based code summarization.

2 CHATGPT FOR AUTOMATIC CODE SUMMARIZATION

In this section, we present five carefully crafted heuristic questions for exploring the performance of ChatGPT [1] in code summarization. The five heuristic questions are as follows:

Q1. Can ChatGPT perform code summarization tasks?
Q2. What does the comment generated by ChatGPT look like?
Q3. How to use ChatGPT to generate concise comments?
Q4. What type of prompt does ChatGPT suggest for generating short comments?
Q5. Which one performs better, the ChatGPT-suggested prompts in Q4 or the prompt proposed in Q3?

Based on observations of answers (feedback) from ChatGPT, we provide an appropriate prompt that guides ChatGPT to generate in-distribution comments. Detailed questions, answers, and observations are described in the following sections.

2.1 Q1: Can ChatGPT perform code summarization tasks?

In this question, we want to figure out whether ChatGPT can be used for code summarization. Figure 2(a) and (b) show the prompt we used to query ChatGPT and the answer of ChatGPT, respectively.

(a) Prompt for Q1

Yes, I can generate comments for a given programming language function. Please specify which programming language you are using and provide the function name and parameters.
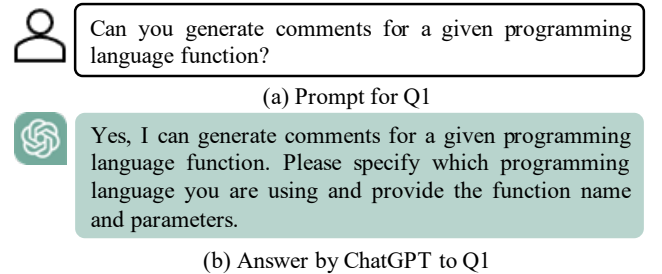
(b) Answer by ChatGPT to Q1

Figure 2: Prompt for Q1 and answer by ChatGPT to Q1

From Figure 2(b), it is observed that ChatGPT clearly answered that it can generate comments for a given programming language function. Therefore, we can conclude that although ChatGPT is not

---
[1]Note that in this paper, we interact with ChatGPT by calling the officially provided API.

primarily intended for automated code summarization, it can still be applied to it.
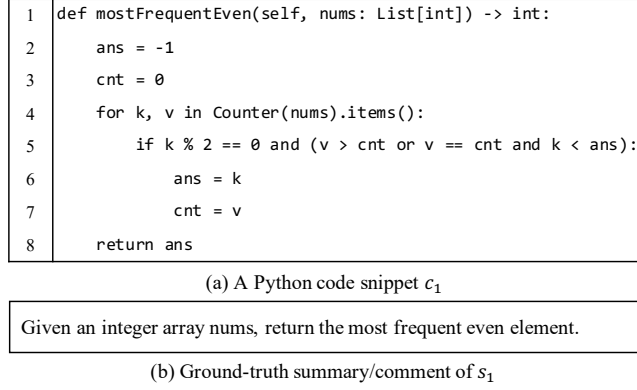
```
1  def mostFrequentEven(self, nums: List[int]) -> int:
2      ans = -1
3      cnt = 0
4      for k, v in Counter(nums).items():
5          if k % 2 == 0 and (v > cnt or v == cnt and k < ans):
6              ans = k
7              cnt = v
8      return ans
```

(a) A Python code snippet c_1

Given an integer array nums, return the most frequent even element.

(b) Ground-truth summary/comment of s_1

Figure 3: An example of code snippet and ground-truth summary/comment

2.2 Q2: What does the comment generated by ChatGPT look like?

In this question, we want to take a look at the comments ChatGPT generates for a specific code snippet. We take the code snippet c_1 in Figure 3(a) as an example and ask ChatGPT to generate comments for it. Figure 4 show the two prompts we used to ask ChatGPT and the corresponding answers by ChatGPT. To avoid unnecessary redundancy, in Figure 4, we use the placeholder "<code>" to refer to the code snippet c_1. From Figure 4(b), it is observed that, compared with the ground-truth comment s_1 shown in Figure 3, the comments generated by ChatGPT for c_1 are very detailed.

Intuitively, different programming languages (e.g., Java and Python) have obvious differences in syntax, style, and coding standards. Therefore, by the way, we verified whether it is possible to not specify a programming language under the premise of providing a code snippet. Figure 4(c) shows an example. Compared with the prompt P1 in Figure 4(a), prompt P2 does not specify the programming language, and the corresponding answer from ChatGPT is shown in Figure 4(d). It is observed that although the prompt P2 does not specify the programming language, the answer to P2 explicitly describes "This is a Python function" in the first line. It means that ChatGPT recognized that c_1 is written in Python. In addition, although there are some differences overall, the core functional summary in the answer to P2 (i.e., "This is a Python function that takes a list of integers as input and returns the most frequent even integer in the list.") is semantically equivalent to that in the answer to P1 (i.e., "This function takes a list of integers as input and returns the most frequently occurring even number in the list."). It indicates that ChatGPT can recognize the programming language of the given code itself, so we do not specify the programming language in the following prompts. Based on the above observations, we can conclude that it is feasible to use ChatGPT for code summarization without specifying the programming language.

2.3 Q3: How to use ChatGPT to generate concise comments?

From the answers to Q2, we observed a phenomenon that the comments generated by ChatGPT are too long, much longer than the ground-truth comment. Usually, we would like to get comments within one sentence. Usually, human developers do not write comments that long either. Therefore, in this question, we explore how to modify the prompts to guide ChatGPT to generate more concise comments.

A naive idea is to add some adjectives with a "concise" meaning to the prompt. Specifically, we tried three adjectives, including "short", "succinct", and "concise". Rows (1)–(3) of Table 1 show the detailed prompts and the corresponding answers by ChatGPT. It is observed that these three comments are significantly shorter than those in Figure 4 (b) and (d), but are still longer than the ground truth (containing 11 words) shown in Figure 3. Compared with the comment generated using the prompt containing "concise" (row (3)), the comment generated using the prompt containing "short" (row (1)) is closer to the ground truth.

2.4 Q4: What kind of prompt does ChatGPT suggest for generating short comments?

In the previous Q3, We found a seemingly good prompt that can instruct ChatGPT to generate short comments. In this question, we want to what kind of prompt ChatGPT suggests for generating short comments. Table 2 lists the four prompts we designed for Q4 and the corresponding answers by ChatGPT.

From rows (1) and (2) of Table 2, it is observed that ChatGPT expects the user prompt to provide a brief description of what the input code snippet does. However, this expectation is what humans expect ChatGPT to do. Therefore, the prompts suggested in the answers of rows (1) and (2) are inappropriate. From rows (3) and (4) of Table 2, it is observed that ChatGPT provides three examples of suggested prompts (highlighted in blue font).

2.5 Q5: Which one performs better, the ChatGPT-suggested prompts in Q4 or the prompt proposed in Q3?

Based on these examples, we design three new prompts. In Table 3, row (1) shows the prompt we find in Q3 and the corresponding answers by ChatGPT, while rows (2)–(4) show the three prompts we designed based on the ChatGPT-suggested prompts in Q4. It is observed that the comments generated using prompts in rows (2)-(4) are significantly longer than that generated using the prompt in row (1).

Further, we use more data to verify the prompt containing "short". Specifically, we randomly select 50 samples from the CSN-Python test set, then use the prompt containing "short" to ask ChatGPT to generate comments. Figure 5 shows the length distribution of the comments generated by ChatGPT using the prompt containing "short" (i.e., ChatGPT (short)) and the ground-truth comments (i.e., Ground-truth) on all 50 samples. It is observed that the median, first quartile, and third quartile associated with ChatGPT (short) are larger than those associated with Ground-truth. To test whether
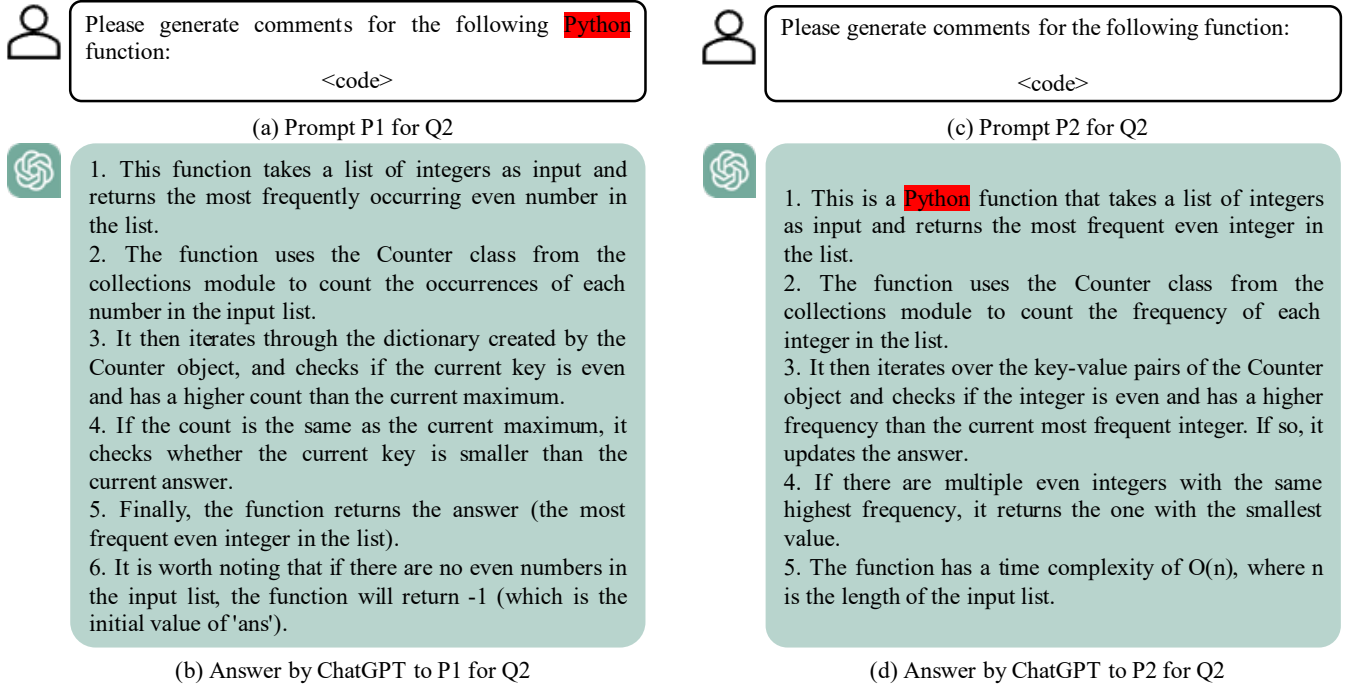
Please generate comments for the following Python function:

<code>

(a) Prompt P1 for Q2

1. This function takes a list of integers as input and returns the most frequently occurring even number in the list.
2. The function uses the Counter class from the collections module to count the occurrences of each number in the input list.
3. It then iterates through the dictionary created by the Counter object, and checks if the current key is even and has a higher count than the current maximum.
4. If the count is the same as the current maximum, it checks whether the current key is smaller than the current answer.
5. Finally, the function returns the answer (the most frequent even integer in the list).
6. It is worth noting that if there are no even numbers in the input list, the function will return -1 (which is the initial value of 'ans').

(b) Answer by ChatGPT to P1 for Q2

Please generate comments for the following function:

<code>

(c) Prompt P2 for Q2

1. This is a Python function that takes a list of integers as input and returns the most frequent even integer in the list.
2. The function uses the Counter class from the collections module to count the frequency of each integer in the list.
3. It then iterates over the key-value pairs of the Counter object and checks if the integer is even and has a higher frequency than the current most frequent integer. If so, it updates the answer.
4. If there are multiple even integers with the same highest frequency, it returns the one with the smallest value.
5. The function has a time complexity of O(n), where n is the length of the input list.

(d) Answer by ChatGPT to P2 for Q2

Figure 4: Two prompts for Q2 and the corresponding answers by ChatGPT to Q2. The "<code>" in the figure represents a placeholder, and the code snippet we filled in during the experiment is c_1 shown in Figure 6(a).

Table 1: Prompts for Q3 and the corresponding answers by ChatGPT. #Words: number of words.

No.	Prompt	Answer	#Words
(1)	Please generate a **short** comment for the following function: <code>	This function finds the most frequent even number in a given list of integers using Python's built-in Counter function.	19
(2)	Please generate a **succinct** comment for the following function: <code>	This function finds the most frequent even number in a given list of integers. It utilizes the Counter function and iterates over the items to find the even number with the highest frequency.	33
(3)	Please generate a **concise** comment for the following function: <code>	This function returns the most frequent even integer from a given list of integers using Counter and basic logic.	19

there is a statistically significant difference between the two methods, we perform the paired Wilcoxon-Mann-Whitney signed-rank test at a significance level of 5%, following previously reported guidelines for inferential statistical analysis involving randomized algorithms [3]. The p-value between ChatGPT (short) and Ground-truth is smaller than the significant threshold value of 0.05, which means the comments generated by ChatGPT (short) are out-of-distribution.

We design two new prompts by adding additional constraints on the prompt containing the "short". Specifically, we use two types of constraints: sentence limitation and word limitation. We first analyze the data distribution of the ground-truth comments in the CSN-Python test set.

Figure 6 shows the statistics of the number of comments according to the number of sentences. It is observed that most ground-truth comments (about 90%) only contain one sentence.

Figure 7 shows the statistics of the number of comments according to the number of words they contained. It is observed that most ground-truth comments (about 99%) consist of less than 40 words.

Row (1) of Table 4 shows the prompt equipped with the sentence limitation (i.e., "in one sentence" highlighted in red font) and the corresponding comment generated by ChatGPT. Rows (2)–(4) show the prompts equipped with the word limitation (i.e., "within/in/no more than 40 words" highlighted in red font) and the corresponding comments generated by ChatGPT. Although we explicitly require ChatGPT to generate comments of less than 40 words, it did not understand it.

Table 2: Prompts for Q4 and the corresponding answers by ChatGPT

No.	Prompt	Answer
(1)	If I give you a programming language function and ask you to generate a short comment for it, how would you like me to raise the question?	You could present the programming language function followed by **a brief explanation of what it does**, and then ask for a concise comment to describe it. For example: "Here is a JavaScript function that generates a random number between 1 and 10. Can you provide a short comment for it?"
(2)	If I give you a programming language function and ask you to generate a short comment for it, how would you like me to prompt?	You can prompt by providing the name of the function and **a brief description of what it does**. Additionally, any input parameters and their descriptions would also be helpful to generate a more specific comment for the function.
(3)	If I **only** give you a function and ask you to generate a short comment for it, how would you like me to raise the question?	You can simply provide the function and ask for a brief description or comment on what the function does or how it works. For example, "**Can you give a short comment on what the reduce() function does in Python?**"
(4)	If I ask you to generate a short comment for a programming language function I give you, how would you like me to phrase the question?	You could ask me to provide a short comment for a specific programming language function by specifying the programming language and the function name. For example, "**Could you generate a short comment for the Python built-in function 'sorted()'?**" or "**What's a concise description of the JavaScript function 'Array.prototype.map()'?**"

Table 3: Prompts and answers by ChatGPT for Q5

| No.	Prompt	Answer
(1)	Please generate a short comment for the following function: <code>	This function finds the most frequent even number in a given list of integers using Python's built-in Counter function.
(2)	Can you give a short comment on what the mostFrequentEven() function does in Python? <code>	The mostFrequentEven() function takes a list of integers nums as input and returns the most frequent even number in the list. It uses the Counter function from the Python collections module to count the occurrences of each element in the list. The function then iterates through the counts and checks if the key (i.e., the number) is even and if its count is greater than the current maximum count or if the count is equal to the maximum count and the key is smaller than the current answer. If so, the function updates the answer and maximum count. Finally, the function returns the answer, which is the most frequent even number in the list. If there are no even numbers in the list, the function returns -1.
(3)	Could you generate a short comment for the Python function 'mostFrequentEven()' <code>	The mostFrequentEven() function in Python takes a list of integers as input and returns the most frequent even number in the list. It uses the Counter() function to count the occurrences of each element in the list and iterates through the counts to find the most frequent even number. If there are no even numbers in the list, the function returns -1.
(4)	What's a concise description of the Python function 'mostFrequentEven()'? <code>	mostFrequentEven() is a Python function that takes a list of integers as input and returns the most frequent even number in the list. It uses the Counter() function to count the occurrences of each element in the list and iterates through the counts to find the most frequent even number. If there are no even numbers in the list, the function returns -1.

Further, we use the randomly selected 50 mentioned earlier to verify the prompts containing "in one sentence" and "within 40 words". Figure 8 shows the length distribution of comments generated by ChatGPT using three prompts, where ChatGPT (within 40 words) and ChatGPT (in one sentence) denote that we use the prompts shown in rows (1)–(2) of Table 4 to guide ChatGPT to generate comments, respectively. From Figure 8, it is observed that except for the comments generated using ChatGPT (one sentence), the length of the comments generated by ChatGPT (within 40 words) and ChatGPT (short) are significantly different from that of the ground-truth comments.

We report and compare the quality of the comments generated by ChatGPT (short), ChatGPT (one sentence), and ChatGPT (within 40 words) on all 50 samples. Table 5 shows the performance of Chat-GPT with the above three prompts. From this table, it is observed that ChatGPT (one sentence) achieves the highest scores of 10.66 in

Table 4: Prompts and the corresponding answers by ChatGPT. #Words: number of words.

No.	Prompt	Answer	#Words
(1)	Please generate a short comment **in one sentence** for the following function: <code>	This function returns the most frequent even number in a given list of integers.	14
(2)	Please generate a short comment **within 40 words** for the following function: <code>	This function finds and returns the most frequent even integer from a given list of integers. It accomplishes this by utilizing the Counter library and iterating through the list to compare the frequency of even integers. The function returns -1 if there are no even integers in the given list.	50
(3)	Please generate a short comment **in 40 words** for the following function: <code>	This function takes in a list of integers and returns the most frequent even integer. It achieves this by using a counter and iterating through the items while checking for even numbers and comparing their frequency. The function is concise and efficient.	42
(4)	Please generate a short comment of **no more than 40 words** for the following function: <code>	This Python function takes a list of integers as input and returns the most frequent even number in the list. It uses the Counter module to count the frequency of each number and compares their counts to determine the most frequent even number.	43
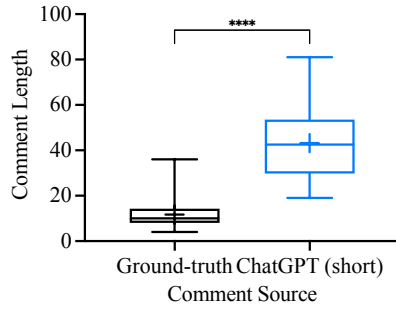
Figure 5: Length of comments generated by ChatGPT and the ground-truth comments. '***' ($p < 0,0001$) represents the differences between the two groups are extremely significant.**
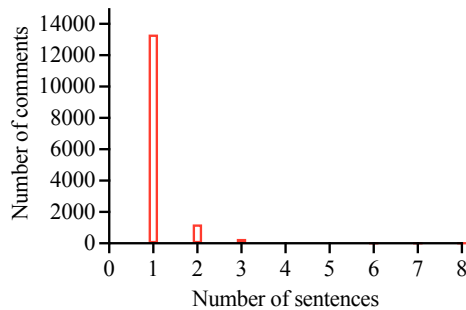
Figure 6: Statistics of the number of sentences in comments
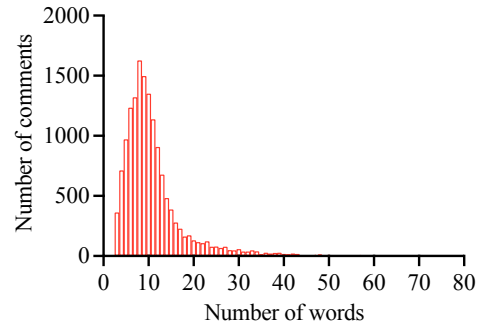
Figure 7: Statistics of the number of words in comments
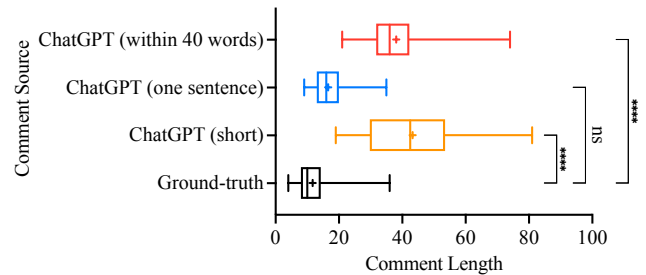
Figure 8: Length distribution of comments on 50 samples. "*" ($0.01 < p < 0.05$), "*" ($0.001 < p < 0.01$), "****" ($0.0001 < p < 0.001$) and "*****" ($p < 0.0001$) represent the differences between two groups are Significant, Very significant, Extremely significant and Extremely significant, respectively. And 'ns' ($p \geq 0.05$) means Not significant.**

BLEU, 13.36 in METEOR, and 20.37 in ROUGE-L, and outperforms ChatGPT (short) and ChatGPT (within 40 words).

Summary.
Based on the carefully crafted heuristic questions and ChatGPT's feedback (including the preliminary results on the randomly selected 50 samples), we find an appropriate prompt that can be used to guide ChatGPT to generate in-distribution comments. The appropriate prompt is "Please generate a short comment in one sentence for the following function:<code>".

Table 5: Performance of ChatGPT with different prompts

Method	CSN-Python		
	BLEU	METEOR	ROUGE-L
ChatGPT (short)	4.85	12.90	14.41
ChatGPT (one sentence)	10.66	13.36	20.37
ChatGPT (within 40 words)	4.83	13.01	14.05

3　EXPERIMENTAL DESIGN

3.1　Dataset

We conduct experiments on a Python dataset provided by the Code-SearchNet (CSN) corpus [29]. Lu et al. [38] showed that some comments in the CSN corpus contain content unrelated to the code snippets and performed data cleaning on the CSN corpus. Therefore, in this paper, we follow [15, 61] and use the clean version of the CSN corpus provided by Lu et al. [38], where the training/validation/test set contains 251,820/13,914/14,918 samples, respectively.

3.1.1　Evaluation Metrics. We use three metrics BLEU [42], METEOR [4], and ROUGE-L [35], to evaluate the quality of comments generated by different models, which are widely used in code summarization [17, 27, 30, 57, 58, 64].

BLEU, the abbreviation for BiLingual Evaluation Understudy [42], is widely used for evaluating the quality of generated code summaries [27, 30, 58]. It is a variant of precision metric, which calculates the similarity by computing the n-gram precision of a generated summary to the reference summary, with a penalty for the overly short length [42]. It is computed as:

$$BLEU = BP * exp(\sum_{n=1}^{N} w_n log p_n) \tag{1}$$

$$BP = \begin{cases} 1, & \text{if } |g| > |r| \\ e^{(1-\frac{|r|}{|g|})}, & \text{if } |g| \leq |r| \end{cases} \tag{2}$$

where $N = 1, 2, 3, 4$ and $w_n = \frac{1}{N}$. p_n is the n-gram precision [48]. BP represents the brevity penalty. g and r denote a generated (predicted) summary and a reference summary, respectively. $|g|$ and $|r|$ denote the lengths of g and r, respectively. In this paper, we follow [17, 64] and show the standard BLEU score which provides a cumulative score of 1-, 2-, 3-, and 4-grams [9].

METEOR, the abbreviation for Metric for Evaluation of Translation with Explicit ORdering [4], is also widely used to evaluate the quality of generated code summaries [59, 66, 68]. For a pair of summaries, METEOR creates a word alignment between them and calculates the similarity scores. Suppose m is the number of mapped unigrams between the reference summary r and the generated summary g, respectively. Then, precision (P_{unig}), recall (P_{unig}), and METEOR are computed as:

$$P_{unig} = \frac{m}{|g|}, \quad R_{unig} = \frac{m}{|r|} \tag{3}$$

$$METEOR = (1 - \gamma * frag^\beta) * \frac{P_{unig} * R_{unig}}{\alpha * P_{unig} + (1 - \alpha) * R_{unig}} \tag{4}$$

where $frag$ is the fragmentation fraction. As in [68], α, β, and γ are three penalty parameters whose default values are 0.9, 3.0 and 0.5, respectively.

ROUGE-L. ROUGE is the abbreviation for Recall-oriented Understudy for Gisting Evaluation [35]. ROUGE-L, a variant of ROUGE, is computed based on the longest common subsequence (LCS). ROUGE-L is also widely used to evaluate the quality of generated code summaries [6, 34, 46]. Specifically, the LCS-based F-measure (F_{lcs}) is called ROUGE-L [35], and F_{lcs} is computed as:

$$R_{lcs} = \frac{LCS(r,g)}{|r|}, \quad P_{lcs} = \frac{LCS(r,g)}{|g|} \tag{5}$$

$$F_{lcs} = \frac{(1 + \beta^2)R_{lcs}P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{6}$$

where r and g also denote the reference summary and the generated summary, respectively. Notice that ROUGE-L is 1 when $g = r$; while ROUGE-L is 0 when $LCS(r,g) = 0$, i.e., which means r and g are completely different. β is set to 1.2 as in [58, 60, 68].

The scores of BLEU, METEOR, and ROUGE-L are in the range of [0,1] and usually reported in percentages. The higher the scores, the closer the generated summary is to the reference summary, and the better the code summarization performance. All scores are computed by the same implementation provided by [68].

3.2　Baseline

In this section, we introduce several DL-based code summarization models selected as baselines, including NCS [1], CodeBERT [15], and CodeT5 [61].

NCS. NCS [1] is the first Transformer-based code summarization model. It adopts a Transformer-based encoder-decoder architecture. It incorporates the copying mechanism [45] in the Transformer to allow both generating words from vocabulary and copying from the source code.

CodeBERT. CodeBERT [15] is a representative pre-trained model for source code. It uses the same model architecture as RoBERTa-base [37]. CodeBERT is trained with the Masked Language Modeling (MLM) task and the Replaced Token Detection (RTD) task. The authors of CodeBERT fine-tune and test it on the code summarization task (also called the code documentation generation task in their paper).

CodeT5. CodeT5 [61] is the SOTA pre-trained model for source code. CodeT5 builds on an encoder-decoder framework with the same architecture as T5 [44]. It is trained with four pre-training tasks, including Masked Span Prediction (MSP) task, Identifier Tagging (IT), Masked Identifier Prediction (MIP), and Bimodal Dual Generation (BDG). Different from CodeBERT, CodeT5 has a pre-trained decoder. The authors of CodeT5 also conduct experiments on the code summarization task.

ChatGPT. Currently, ChatGPT [41] is the most advanced and possibly the most powerful Large Language Model (LLM). It is trained with massive texts and codes, including a wide range of user queries, and responses that are coherently relevant to these queries. These training data make ChatGPT applicable to chatbots or virtual assistants. Furthermore, it also employs advanced supervised instruction fine-tuning techniques and RLHF [10] to adapt

more effectively to specific tasks or domains. In this paper, we conducted experiments to investigate ChatGPT's capability on code summarization tasks.

4 RESULT ANALYSIS AND CASE STUDY

4.1 Result Analysis

Table 6: Performance of ChatGPT and three SOTA models

Method	CSN-Python		
	BLEU	METEOR	ROUGE-L
NCS	15.8	10.6	31.3
CodeBERT	18.7	12.4	34.8
CodeT5	20.0	14.7	37.7
ChatGPT (one sentence)	10.28	14.40	20.81

In this section, we report and compare the comments generated by ChatGPT (one sentence) and three SOTA models (including NCS, CodeBERT, and CodeT5) on the entire test set of the CSN-Python dataset. ChatGPT (one sentence) denotes that we uniformly use the following prompt to guide ChatGPT to generate comments for all code snippets in the CSN-Python test set:

Please generate a short comment in one sentence for the following function:
<code>

Table 6 shows the overall performance of ChatGPT (one sentence), NCS, CodeBERT, and CodeT5. From this table, it is observed that 1) in terms of BLEU and ROUGE-L, ChatGPT obtains the lowest scores of 10.28 in BLEU and 20.81 in ROUGE-L, worse than all three SOTA baselines; in terms of METEOR, ChatGPT is comparable to CodeT5 and is better than NCS and CodeBERT.
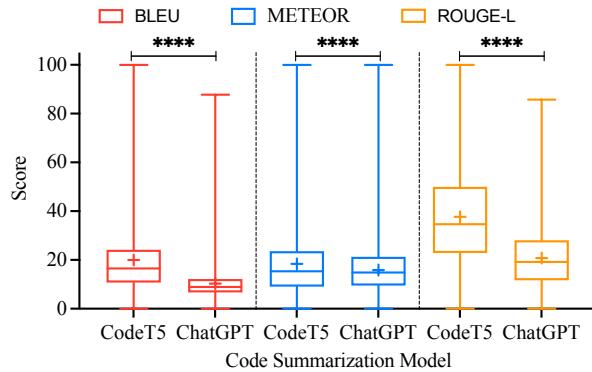

Figure 9: Distribution of metric scored obtained by CodeT5 and ChatGPT (one sentence)

Figure 9 shows the score distribution of the comments generated by ChatGPT and the best baseline CodeT5 on all samples in the test set of the CSN-Python dataset. It is observed that 1) in terms of BLEU and ROUGE-L, the median, first quartile, and third quartile associated with CodeT5 are larger than those associated with ChatGPT; 2) in terms of METEOR, the third quartile associated with CodeT5 is larger than those associated with ChatGPT. To

test whether there is a statistically significant difference between CodeT5 and ChatGPT, we perform the paired Wilcoxon-Mann-Whitney signed-rank test at a significance level of 5%. All p-values between CodeT5 and ChatGPT for all three metrics are smaller than the significant threshold value of 0.05, which means the comments generated by CodeT5 are significantly better than that generated by ChatGPT.

4.2 Case Study

In this section, we provide case studies to understand the generated summaries of ChatGPT compared with CodeT5 to demonstrate the strengths and weaknesses of ChatGPT

4.2.1 successful cases. Figure 10(a) is a code snippet in CSN-Python and Figure 10(b) shows the reference summary and the comments generated by CodeT5 and ChatGPT. The comment generated by ChatGPT cover the main semantics of "Establish a connection" and "to druid broker" in ground-truth. But the comment generated by CodeT5 use the wrong verb "Get" instead of "Establish", which may be caused by the frequent appearance of "get connection" in the code. It indicats that ChatGPT can better understand the inner meaning of the code without being interfered by the vocabulary of the code. Besides, ChatGPT tends to describe more detailed behavior of the code in the comments generated, such as "returns the connection object".

```
def get_conn(self):
    conn = self.get_connection(self.druid_broker_conn_id)
    druid_broker_conn = connect(
    host=conn.host,
    port=conn.port,
    path=conn.extra_dejson.get('endpoint', '/druid/v2/sql'),
        scheme=conn.extra_dejson.get('schema', 'http')
    )
    self.log.info('Get the connection to druid broker on %s',
conn.host)
    return druid_broker_conn
```

(a) A code snippet c_2

Reference Summary: Establish a connection to druid broker .
CodeT5: Get the connection to druid broker
ChatGPT: is used to establish a connection to a Druid broker, and returns the connection object.

(b) Comments generated by CodeT5 and ChatGPT for c_2

Figure 10: A successful example of ChatGPT

4.2.2 moderate cases. Figure 11(b) shows the reference summary and the comments generated by CodeT5 and ChatGPT for the code snippet in Figure 11(a) (a code snippet in CSN-Python). The comments generated by CodeT5 and ChatGPT both cover the main semantics of "Print a log message" in the reference summary. However, the second part "to standard error" is ignored in the comments generated by ChatGPT and is replaced irrelevant information "with color coding". It is worth mentioning that the comment generated by CodeT5 contains "to stderr", which might have taken a hint from "sys.stderr.write" in the code. It demonstrates that, compared with CodeT5, ChatGPT is less sensitive to the features or keywords of a programming language.
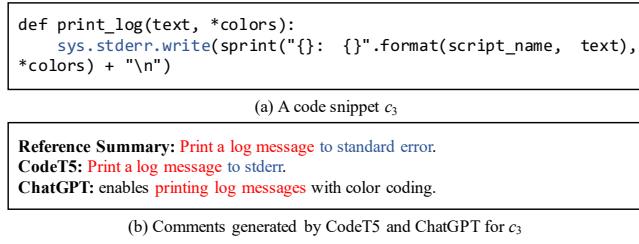
```
def print_log(text, *colors):
    sys.stderr.write(sprint("{}: {}".format(script_name, text),
*colors) + "\n")
```

(a) A code snippet c_3

Reference Summary: Print a log message to standard error.
CodeT5: Print a log message to stderr.
ChatGPT: enables printing log messages with color coding.

(b) Comments generated by CodeT5 and ChatGPT for c_3

Figure 11: A moderate example of ChatGPT

4.2.3 weak cases. Figure 12(a) is a code snippet in CSN-Python. Figure 12(b) shows the reference summary and the comments generated by CodeT5 and ChatGPT. Neither CodeT5 nor ChatGPT realized that the function of the code is to "Execute SQL" statements. They misunderstood it for the copy operation, which may be caused by ignoring the sentence "conn.commit()". It reveals that ChatGPT might still have some difficulty understanding the deep logic of the code.
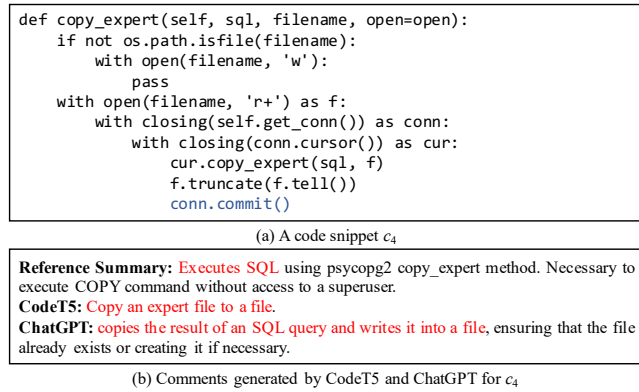
```
def copy_expert(self, sql, filename, open=open):
    if not os.path.isfile(filename):
        with open(filename, 'w'):
            pass
    with open(filename, 'r+') as f:
        with closing(self.get_conn()) as conn:
            with closing(conn.cursor()) as cur:
                cur.copy_expert(sql, f)
                f.truncate(f.tell())
                conn.commit()
```

(a) A code snippet c_4

Reference Summary: Executes SQL using psycopg2 copy_expert method. Necessary to execute COPY command without access to a superuser.
CodeT5: Copy an expert file to a file.
ChatGPT: copies the result of an SQL query and writes it into a file, ensuring that the file already exists or creating it if necessary.

(b) Comments generated by CodeT5 and ChatGPT for c_4

Figure 12: A weak example of ChatGPT

5 RELATED WORK

The neural machine translation (NMT)-based models are widely used to generate summaries for code snippets with encoder-decoder neural networks [17, 18, 25, 26, 30, 32, 34, 64]. For example, Iyer et al. [30] are the first to apply deep learning to automatic code summarization. They adopt LSTM networks [24] with attention to leverage the code vectors and generate natural language sentences that describe C# code snippets and SQL queries. Hu et al. [27] use one additional encoder to encode API sequences and improve the summary generation by learned the API knowledge. Subsequently, various additional information is incorporated to further improve DL-based code summarization performance, such as abstract syntax trees [17, 32, 34, 49, 58, 64], code property graphs [36], similar code snippets [33, 62], file context [22], etc. Recently, with the success of the pre-training and fine-tuning paradigm in the field of NLP (e.g., BERT [12] and T5 [44]), many works introduce this paradigm to further boost neural code summarization, such as CodeBERT [15] and CodeT5 [61]. These works first pre-train a model with general

language modeling tasks, such as MLM and ULM. Then, they fine-tune the pre-trained models on code summarization [15, 19, 61].

Recently, there are growing interest in leveraging ChatGPT for various software engineering tasks. Sobania et al. [51] compare the performance of ChatGPT with the performance of Codex and several dedicated automatic program repair (APR) approaches. They claim that human input can be of much help to an automated APR system, but it requires a mental cost to verify ChatGPT answers. They propose that incorporation of automated approaches to provide ChatGPT with hints as well as automated verification of its responses would yield ChatGPT to be a viable tool. Cao et al. [7] study ChatGPT's ability in DL program debugging. They explore the impact of prompts on the debugging performance and propose a prompt template to improve the performance. Additionally, they explore how far ChatGPT's dialogue feature can push the performance forward. They summarize the pros and cons of ChatGPT's reply and point out the possible directions where ChatGPT could help to facilitate the SE community. Tian et al. [55] investigate ChatGPT on three code-related tasks: code generation, program repair, and code summarization. They find out that ChatGPT struggles to generalize to new and unseen problems and long prompts have a negative impact on the inference capabilities of ChatGPT. They also discover that ChatGPT has a limited attention span.

6 THREATS TO VALIDITY

Threats to External Validity. ChatGPT generate varied answers for identical code snippets with different prompts due to their inherent randomness. One threat to the validity of our study is that conclusions drawn from random results may be misleading. To mitigate this threat, we conducted a pre-study to determine the prompt so that the comments generated by ChatGPT are more like real-life comments. In addition, we conduct our experiment on a large dataset, which helps reduce the impact of randomness to some extent. Another potential threat to validity is the selection of the programming language and benchmark dataset. The results might vary depending on the programming language and size of the benchmark dataset. To mitigate these threats, we chose a popular large benchmark dataset (i,e., the CSN corpus) and targeted Python – the most popular programming language. We will explore the code summarization performance of ChatGPT on other programming language datasets provided in the CodeSearchNet corpus in future work.

Threats to Construct Validity. ChatGPT is currently under active development. The ChatGPT we evaluate is based on the GPT-3.5 model. During our study there was a major update to it, the latest model is GPT-4. Therefore, the results in this paper might have been an underestimation of the latest ChatGPT. In future studies, we plan to investigate the GPT-4-based ChatGPT and compare its performance with that of previous versions.

7 OPPORTUNITIES

(1) A lot of room for improvement on existing datasets.
In this paper, we directly run ChatGPT on the CSN-Python test set without any fine-tuning process. This is a zero-shot application scenario. From the results in Table 6, it is observed that Chat-GPT's zero-short code summarization performance still has a lot of

Please generate a short comment for the following function:
```
def validate_value(key, value, expected_type):
    if not isinstance(valueexpected type):
        raise TypeError("{} argument must have a type{} not {}".format(
            key, expected_type, type(value)))
```

This is a Python function that validates the type of a given value against an expected type. The function takes three arguments: key, value, and expected_type. The key argument is a string that identifies the value being validated, the value argument is the actual value being validated, and the expected_type argument is the type that the value is expected to be.

The function works by using the isinstance() function to check if the value argument is an instance of the expected_type. If the value is not an instance of the expected type, the function raises a TypeError with a message that describes the error.

The error message includes the key argument, the expected_type argument and the type of the value argument. This provides a clear message to the caller of the function about what went wrong and what they need to do to fix the error.

Overall, this function is useful for enforcing type constraints in Python code and can help catch errors early in the development process.

(a) An example of rich but lengthy comments generated by ChatGPT

GT function to check expected type and raise error if type is not correct.

(b) The ground-truth comment of the function "validate_value()" in (a)

```
def validate_value(key, value, expected_type):
    """ Validate the type of a given value against an expected type.
    key: a string that identifies the value being validate;
    value: is the actual value being validated;
    expected_type: the type that the value is expected to be.
    """
    if not isinstance(valueexpected type):
        # If the value is not an instance of the expected type, the
        function raises a TypeError with a message that describes the error.
        raise TypeError(
            "{} argument must have a type{} not {}".format(
            key, expected_type, type(value))
        )
```

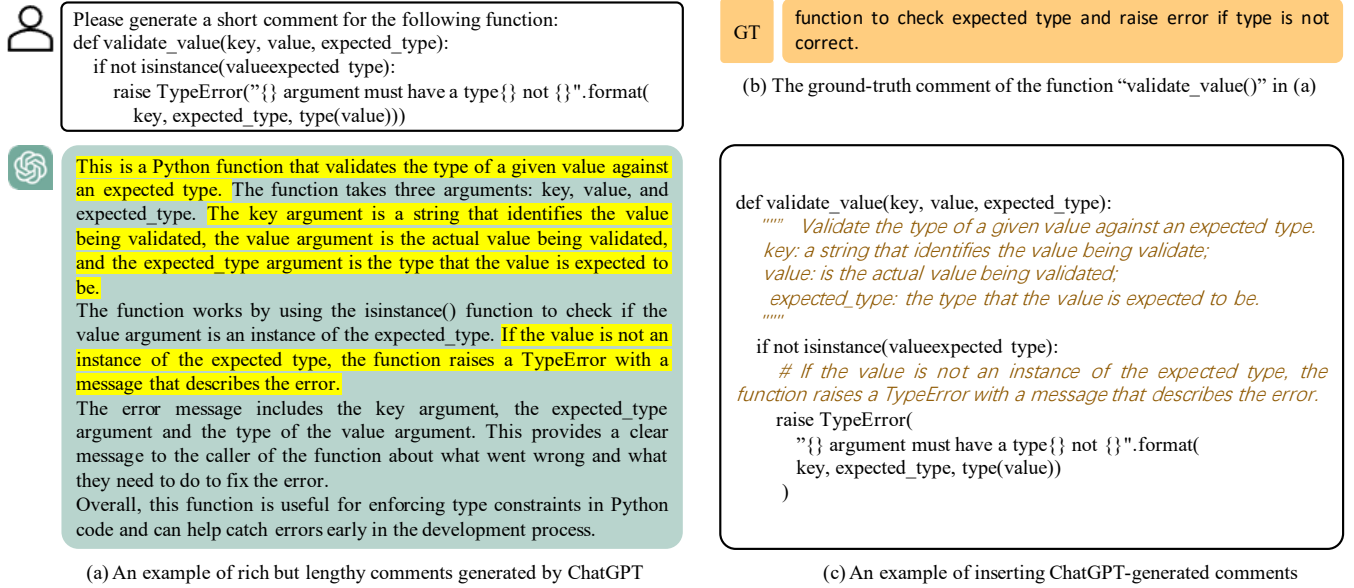(c) An example of inserting ChatGPT-generated comments

Figure 13: An example of cropping and templating comments generated by ChatGPT

room for improvement on the CSN-Python dataset. Since the large language model GPT 3.5/4 behind ChatGPT is not open-sourced, fine-tuning it is not practical. A plausible way to improve its code summarization performance is to explore more efficient prompts. However, it is time-consuming and labor-intensive to rely on manual attempts. How to use prompt engineering to automatically explore better prompts is worthy of further study. In this paper, we provide two optimization directions to explore appropriate prompts to guide ChatGPT to generate in-distribution comments, i.e., sentence limitation and word limitation. Other optimization directions are also worthy of further exploration.

(2) Cropping and templating the rich but lengthy comments that ChatGPT generates by default.

In Section 2, we find that ChatGPT can generate rich and lengthy comments. It should be noted that although the comments generated by ChatGPT are relatively long, it does not mean that these long comments are useless. Figure 13 shows an example, where (a) shows the prompt and comments generated by ChatGPT for the function "validate_value()"; (b) shows the ground-truth comment of the function "validate_value()"; and (c) shows the example of inserting the cropping and templating comments generated by ChatGPT into the function "validate_value()". From the figure, it is observed that 1) compared with the ground-truth comment, the comments generated by ChatGPT contain more details; 2) cropping the comments generated by ChatGPT and wrapping the cropped comments (e.g., the comments highlighted in yellow shown in Figure 13(a)) with a template (e.g., the docstrings inserted in Figure 13(c)) are very useful and friendly for developers. Therefore, how to automatically extract concise and useful information from such comments is also worthy of further study. Python enhancement proposals [56] provide detailed coding conventions, which can be used to guide the design of the comment template.

(3) Creating a new benchmark dataset.

Please generate a short comment for the following function:
```
def vimeo_download_by_channel(url, output_dir='.'
    , merge=False, info_only=False, **kwargs):
    channel_id = match1(url, r'http://vimeo.com/channels/(\w+)')
    vimeo_download_by_channel_id(channel_id, output_dir
    , merge, info_only, **kwargs)
```

(a) A code snippet c_5

This Python function downloads Vimeo videos by channel ID.

(b) The comment generated by ChatGPT for c_5

GT str - > None.

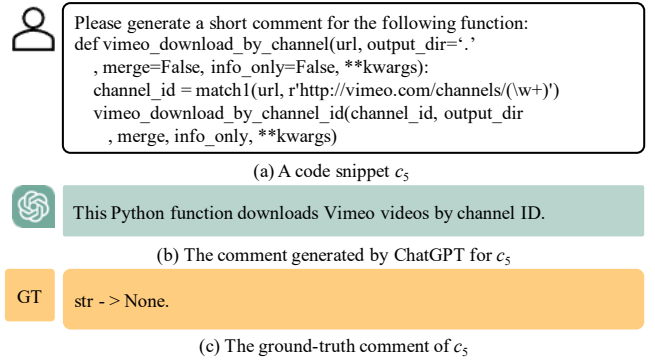(c) The ground-truth comment of c_5

Figure 14: An example of a low-quality ground-truth comment

Currently, the quality of datasets (e.g., the CSN corpus [29]) widely used in code summarization is unverified. These data sets are usually crawled from open-source platforms (such as GitHub [16]) and then subjected to simple data cleaning. We find some ground-truth comments are low-quality. Figure 14 shows an example of a low-quality ground-truth comment. From Figure 14(c), it is observed that the ground-truth comment of c_5 is too short and does not summarize the functionality of c_5. Compared with the ground-truth comment, the comment generated by ChatGPT for c_5 is more informative and summarizes the core semantics of c_5. Therefore, we believe that how to leverage ChatGPT to construct a high-quality benchmark dataset for code summarization is worth further exploration.

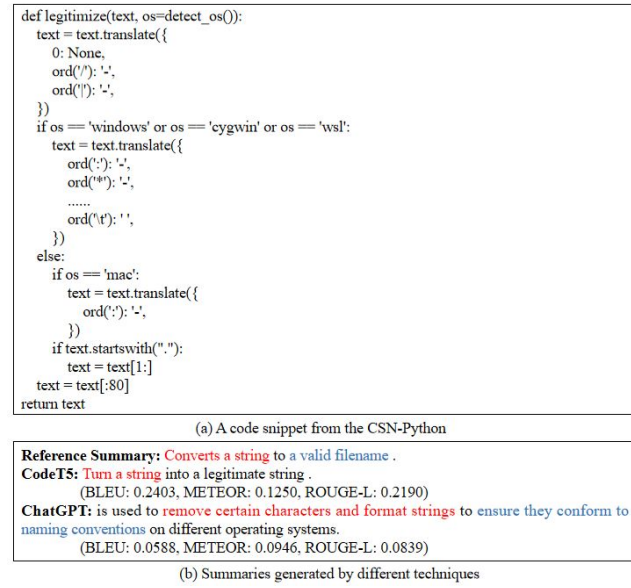**(4) Designing new metrics to evaluate code summarization models.**

```
def legitimize(text, os=detect_os()):
    text = text.translate({
        0: None,
        ord('/'): '-',
        ord('|'): '-',
    })
    if os == 'windows' or os == 'cygwin' or os == 'wsl':
        text = text.translate({
            ord(':'): '-',
            ord('*'): '-',
            ......
            ord('\t'): ' ',
        })
    else:
        if os == 'mac':
            text = text.translate({
                ord(':'): '-',
            })
        if text.startswith("."):
            text = text[1:]
    text = text[:80]
    return text
```

(a) A code snippet from the CSN-Python

Reference Summary: Converts a string to a valid filename .
CodeT5: Turn a string into a legitimate string .
    (BLEU: 0.2403, METEOR: 0.1250, ROUGE-L: 0.2190)
ChatGPT: is used to remove certain characters and format strings to ensure they conform to naming conventions on different operating systems.
    (BLEU: 0.0588, METEOR: 0.0946, ROUGE-L: 0.0839)

(b) Summaries generated by different techniques

Figure 15: An example of traditional metrics not suitable for evaluating the quality of the comments generated by Chat-GPT

Figure 15(a) shows a code snippet c_6, and Figure 15(b) shows the ground-truth comment and the comments generated by CodeT5 and ChatGPT. From the figure, it is observed that although the comment generated by CodeT5 outperforms the one generated by ChatGPT in BLEU, METEOR and ROUGE-L, the semantic "a valid filename" is missing in the comment generated by CodeT5. However, the comment generated by ChatGPT covers the semantics of "Converts a string" and "to a valid filename" though not using the exact words. This is strong evidence that traditional evaluation metrics are no longer suitable for evaluating the quality of the comments generated by ChatGPT.

8 CONCLUSION

In this paper, we preliminarily design heuristic questions to explore an appropriate prompt that can guide ChatGPT to generate in-distribution comments. We compared the automatic code summarization performance of ChatGPT with that of three dedicated code summarization models (i.e., NCS, CodeBERT, and CodeT5) on the CSN-Python dataset. We find that overall ChatGPT is inferior to dedicated code summarization models on the CSN-Python dataset in terms of BLEU, METEOR, and ROUGE-L. Based on our findings, we outline four open challenges and opportunities for applying ChatGPT to code summarization. We hope our results and observations will be helpful for future work with ChatGPT in the field of code summarization.

REFERENCES

[1] Wasi Uddin 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*. Association for Computational Linguistics, Online, 4998–5007.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations-Poster*. OpenReview.net, New Orleans, LA, USA, 1–13.

[3] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[4] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Association for Computational Linguistics, Ann Arbor, Michigan, USA, 65–72.

[5] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. 2023. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023* (2023).

[6] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-Level Encoding for Neural Source Code Summarization of Subroutines. In *Proceedings of the 29th International Conference on Program Comprehension*. IEEE, Madrid, Spain, 253–264.

[7] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. *arXiv preprint arXiv:2304.08191* (2023).

[8] Jialun Cao, Meiziniu Li, Ming Wen, and Shing chi Cheung. 2023. A study on Prompt Design, Advantages and Limitations of ChatGPT for Deep Learning Program Repair. *arXiv preprint arXiv:2304.08191* (2023).

[9] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. *ACM Transactions on Software Engineering and Methodology* 30, 2 (2021), 25:1–25:29.

[10] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep Reinforcement Learning from Human Preferences. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems*. Curran Associates Inc., Long Beach, CA, USA, 4299–4307.

[11] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia Marçal de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: documenting & Designing for Pervasive Information*. ACM, Coventry, UK, 68–75.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 23th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Minneapolis, MN, USA, 4171–4186.

[13] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *arXiv preprint arXiv:2304.07590* (2023).

[14] Yunhe Feng, Sreecharan Vanam, Manasa Cherukupally, Weijian Zheng, Meikang Qiu, and Haihua Chen. 2023. Investigating Code Generation Performance of ChatGPT with Crowdsourcing Social Data. In *Proceedings of the 47th IEEE Computer Software and Applications Conference*. 1–10.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings*. Association for Computational Linguistics, Online Event, 1536–1547.

[16] Inc. GitHub. 2008. GitHub. site: https://github.com. Accessed May, 2023.

[17] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source Code Summarization with Structural Relative Position Guided Transformer. *CoRR* abs/2202.06521 (2022).

[18] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": data, metrics, Baselining & Evaluation. In *Proceedings of the 35th International Conference on Automated Software Engineering*. IEEE, Melbourne, Australia, 746–757.

[19] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: unified Cross-Modal Pre-training for Code Representation. *CoRR* abs/2203.03850 (2022), 1–14.

[20] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd International Conference on Software Engineering*. ACM, Cape Town, South Africa, 223–226.

[21] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *Proceedings of the 17th Working Conference on Reverse Engineering*. IEEE Computer Society, Beverly, MA, USA, 35–44.

[22] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved Automatic Summarization of Subroutines via Attention to File Context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, Seoul, Republic of Korea, 300–310.

[23] Carl S. Hartzman and Charles F. Austin. 1993. Maintenance productivity: observations based on an experience in a large system environment. In *Proceedings of the 3rd Conference of the Centre for Advanced Studies on Collaborative Research.* IBM, Toronto, Ontario, Canada, 138–170.

[24] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[25] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th International Conference on Program Comprehension.* ACM, Gothenburg, Sweden, 200–210.

[26] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep Code Comment Generation with Hybrid Lexical and Syntactical Information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence.* ijcai.org, Stockholm, Sweden, 2269–2275.

[28] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' Expectations on Automated Code Comment Generation. In *Proceedings of the 44th International Conference on Software Engineering.* ACM, Pittsburgh, PA, USA, 1693–1705.

[29] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019).

[30] 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.* The Association for Computer Linguistics, Berlin, Germany, 2073–2083.

[31] Wenxiang Jiao, Wenxuan Wang, Jen tse Huang, Xing Wang, and Zhaopeng Tu. 2023. Is ChatGPT a good translator? A preliminary study. *arXiv preprint arXiv:2301.08745* (2023).

[32] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering.* IEEE / ACM, Montreal, QC, Canada, 795–806.

[33] Jia Li, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. 2021. EditSum: a Retrieve-and-Edit Framework for Source Code Summarization. In *Proceedings of the 36th International Conference on Automated Software Engineering.* IEEE, Melbourne, Australia, 155–166.

[34] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting. In *Proceedings of the 29th International Conference on Program Comprehension.* IEEE, Madrid, Spain, 184–195.

[35] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics – workshop on Text Summarization Branches Out.* Association for Computational Linguistics, Barcelona, Spain, 74–81.

[36] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Automatic Code Summarization via Multi-dimensional Semantic Fusing in GNN. *CoRR* abs/2006.05405 (2020).

[37] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019), 1–13.

[38] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks.* virtual, 1–14.

[39] Paul W. McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, Hyderabad, India, 279–290.

[40] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *Proceedings of the 21st International Conference on Program Comprehension.* IEEE Computer Society, San Francisco, CA, USA, 23–32.

[41] OpenAI. 2015. ChatGPT: Optimizing language models for dialogue. site: https://openai.com/. Accessed: 2023-04-29.

[42] 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.* ACL, Philadelphia, PA, USA, 311–318.

[43] Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. 2023. Is ChatGPT a general-purpose natural language processing task solver? *arXiv preprint arXiv:2302.06476* (2023).

[44] 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 (2020), 1–67.

[45] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get To The Point: Summarization with Pointer-Generator Networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics.* Association for Computational Linguistics, Vancouver, Canada, 1073–1083.

[46] Ramin Shahbazi, Rishab Sharma, and Fatemeh H. Fard. 2021. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In *Proceedings of the 29th International Conference on Program Comprehension.* IEEE, Madrid, Spain, 411–421.

[47] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2021. Neural Code Summarization: How Far Are We?. In *CoRR*, Vol. abs/2107.07112.

[48] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the Evaluation of Neural Code Summarization. In *Proceedings of the 44th International Conference on Software Engineering.* IEEE, Pittsburgh, USA, 1597–−1608.

[49] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic Source Code Summarization with Extended Tree-LSTM. In *Proceedings of the 18th International Joint Conference on Neural Networks.* IEEE, Hungary, 1–8.

[50] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of ChatGPT. *arXiv preprint arXiv:2301.08653* (2023).

[51] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[52] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the 25th International Conference on Automated Software Engineering.* ACM, Antwerp, Belgium, 43–52.

[53] Weisong Sun, Chunrong Fang, Yuchen Chen, Quanjun Zhang, Guanhong Tao, Tingxu Han, Yifei Ge, Yudu You, and Bin Luo. 2022. An Extractive-and-Abstractive Framework for Source Code Summarization. *CoRR* abs/2206.07245 (2022).

[54] Ted Tenny. 1988. Program Readability: Procedures Versus Comments. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1271–1279.

[55] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[56] Guido van Rossum, Barry Warsaw, and Nick Coghlan. 2001. PEP 8 – Style Guide for Python Code. site: https://peps.python.org/pep-0008/#comments. Accessed May, 2023.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems.* Curran Associates Inc., Long Beach, CA, USA, 5998–6008.

[58] 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 International Conference on Automated Software Engineering.* ACM/IEEE, Montpellier, France, 397–407.

[59] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip Yu, and Guandong Xu. 2020. Reinforcement-Learning-Guided Source Code Summarization Using Hierarchical Attention. *IEEE Transactions on Software Engineering (Early Access)* (2020), 1–19.

[60] Yanlin Wang, Ensheng Shi, Lun Du, Xiaodi Yang, Yuxuan Hu, Shi Han, Hongyu Zhang, and Dongmei Zhang. 2021. CoCoSum: Contextual Code Summarization with Multi-Relational Graph Neural Network. *CoRR* abs/2107.01933 (2021), 1–24.

[61] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 26th Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, Virtual Event / Punta Cana, Dominican Republic, 8696–8708.

[62] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and Refine: exemplar-based Neural Comment Generation. In *Proceedings of the 35th International Conference on Automated Software Engineering.* IEEE, Melbourne, Australia, 349–360.

[63] Scott N. Woodfield, Hubert E. Dunsmore, and Vincent Yun Shen. 1981. The Effect of Modularization and Comments on Program Comprehension. In *Proceedings of the 5th International Conference on Software Engineering.* IEEE Computer Society, San Diego, California, USA, 215–223.

[64] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *Proceedings of the Findings of the 59th Annual Meeting of the Association for Computational Linguistics.* Association for Computational Linguistics, Online Event, 1078–1090.

[65] Xianjun Yang, Yan Li, Xinlu Zhang, Haifeng Chen, and Wei Cheng. 2023. Exploring the limits of ChatGPT for query or aspect-based text summarization. *arXiv preprint arXiv:2302.08081* (2023).

[66] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *Proceedings of the 29th International Conference on Program Comprehension.* IEEE, Madrid, Spain, 1–12.

[67] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the 42nd International Conference on Software Engineering.* ACM, Seoul, South Korea, 1359–1371.

[68] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *Proceedings of the 42nd International Conference on Software Engineering.* ACM, Seoul, South Korea, 1385–1397.