Purpose Enhanced Reasoning through Iterative Prompting: Uncover Latent Robustness of ChatGPT on Code Comprehension

Yi Wang, **Qidong Zhao**, **Dongkuan Xu** and **Xu Liu**
North Carolina State University
ywang336@ncsu.edu, qzhao24@ncsu.edu, dxu27@ncsu.edu, xliu88@ncsu.edu

Abstract

Code comments are crucial for gaining in-depth insights to facilitate code comprehension. The key to obtaining these insights lies in precisely summarizing the main purpose of the code. Recent approaches on code comment generation lie in prompting large language models (LLMs) such as ChatGPT, instead of training/fine-tuning specific models. Although ChatGPT demonstrates an impressive performance in code comprehension, it still suffers from robustness challenges in consistently producing high-quality code comments. This is because ChatGPT prioritizes the semantics of code tokens, which makes it vulnerable to commonly encountered benign perturbations such as variable name replacements. This study proposes a modular prompting paradigm *Perthept* to effectively mitigate the negative effects caused by such minor perturbations. *Perthept* iteratively enhances the reasoning depth to reach the main purpose of the code. *Perthept* demonstrates robustness under the scenario where there is stochasticity or unreliability in ChatGPT's responses. We give a comprehensive evaluation across four public datasets to show the consistent robustness improvement with our proposed methodology over other models.

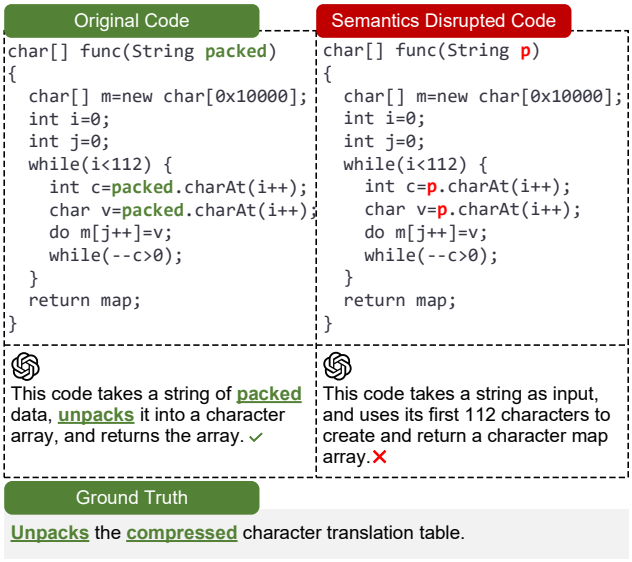


Figure 1: This figure provides a case of the decrease in the quality of code comment generation. The original codes on the left side represent an unpacking algorithm, and the code summary generated by ChatGPT includes the keyword "unpack". When argument name `packed` is transformed into a commonly used name `p`, as shown on the right side, ChatGPT fails to identify that it is an unpacking algorithm. This case indicates the robustness issue of ChatGPT in code comprehension.

1 Introduction

Code comment generation has attracted extensive attention in recent years. High-quality code summaries provide deep insights into code behavior, facilitating code comprehension. Automatic code comment generation has been widely used in program comprehension [Sridhara *et al.*2010], software development [de Souza *et al.*2005], bug fixing [Panichella *et al.*2016], and software maintenance [He2019].

Large language models (LLMs) with exceedingly large training datasets and optimized training approaches have shown remarkable abilities in various generation tasks including code comment generation, showcasing extensive generalization and advanced semantics comprehension ability. Programmers can use a straightforward prompt like "summarize the code" to interact with the LLM and receive a satisfactory code summary. Therefore, there is a growing trend of prompting LLMs (e.g., ChatGPT[1], GPT-4[2]) for generating code summaries. However, LLMs tend to generate code summaries based on the semantic information of code tokens, which results in missing deep insights into code behavior and suffering robustness issues in consistently producing high-quality code summaries when a minor perturbation (e.g., function/variable/argument name replacement) is introduced into the source code. Take ChatGPT as an example, as shown in Figure 1, when the semantic-rich argument name `packed` is replaced with an unrelated but commonly used argument name `p`, ChatGPT losses track of the main purpose of the code, an "unpacking" algorithm, which is crucial for programmers to accurately comprehend the code. Instead,

[1]https://openai.com/blog/chatgpt
[2]https://openai.com/research/gpt-4

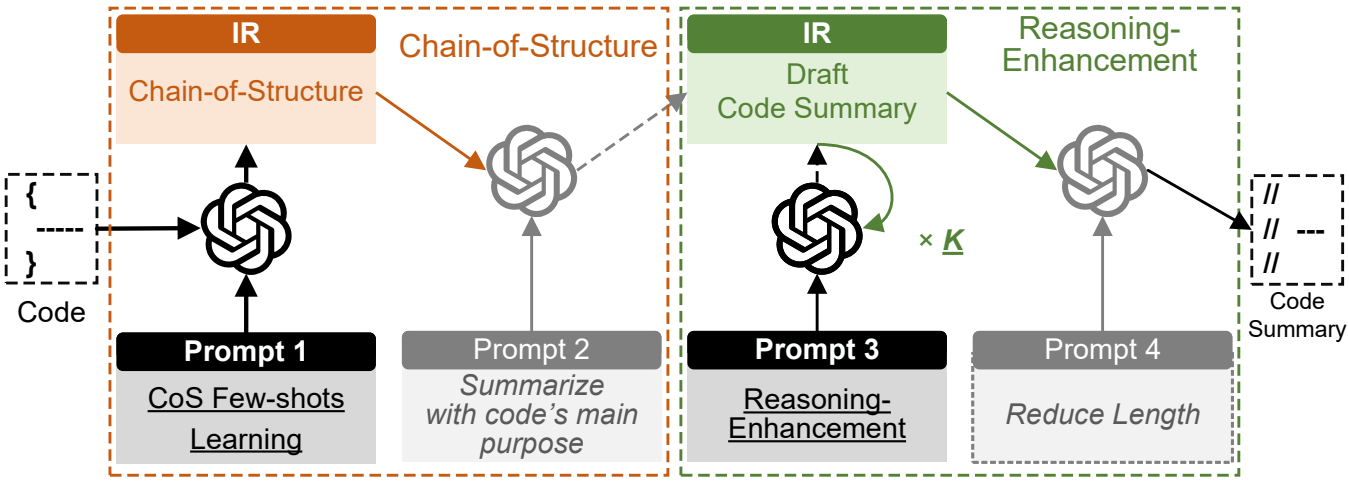


Figure 2: This figure gives an overall workflow of this study. There are four prompts included, and the key prompts are prompt 1 and prompt 3. The orange section is *Chain-of-Structure* generation paradigm, and the green section represents *Reasoning-Enhancement* paradigm. The input source code is initially transformed into a special-grained summarized *Chain-of-Structure* information (using prompt 1). This representation of information is used for iterative reasoning (using prompt 3) to uncover the high-level main purpose of the code. Ultimately, this process leads to the generation of a high-quality code summary.

ChatGPT formattedly generates lengthy and useless descriptions of the code.

Since fine-tuning ChatGPT is costly, prompt design serves as a practical solution to effectively address this robustness issue. We propose *Perthept* (**P**urpose **e**nhanced **r**easoning **th**rough it**e**rative **p**rompt**t**ing), a prompting framework to uncover the latent capability of large language models to generate robust and high-quality code summaries. *Perthept* is composed of two prompting paradigms, *Chain-of-Structure* and *Reasoning-Enhancement*. We develop *Chain-of-Structure* prompting with a special-designed granularity of information detail, that prompts ChatGPT to shift its attention from unreliable semantics information to reliable structure information, facilitating purpose reasoning. Meanwhile, *Chain-of-Structure* helps to ensure no information loss. In addition, we design a novel paradigm *Reasoning-Enhancement* to prompt ChatGPT to perform a self-evaluation of the extracted main purpose of the code. Therefore, ChatGPT can further refine its predictions at a higher reasoning level. The overall workflow is shown in Figure 2. This workflow not only precisely provides deep insights for code summaries, but also mitigates the negative influence caused by the inherent unreliability of ChatGPT. Finally, we evaluate *Perthept* on the TL-CodeSum datasets [Hu *et al.*2018b], the CodeSearchNet Java datasets [Husain *et al.*2019], the CodeSearchNet Python datasets [Husain *et al.*2019], and the Python datasets [Wan *et al.*2018] respectively. In order to exclude samples that do not align with the focus of this study, which is reasoning the high-level main purpose of the code, we develop a filter algorithm to extract the potential reasoning-required code samples from these well-studied datasets because most code samples are simple and do not require high-level reasoning. We summarize the contributions as follows:

- We identify and assess the purpose-reasoning ability of ChatGPT for code comprehension. Extensive experiments show that purpose-reasoning can be effectively utilized to uncover ChatGPT's inherent capacity in producing robust and insightful code summaries.
- We introduce a modular framework, *Perthept*, designed to uncover the latent robustness of ChatGPT by iteratively enhancing the level of reasoning. Comprehensive experiments suggest that *Perthept* demonstrates consistent performance in assisting ChatGPT in producing high-quality code summaries and maintaining robustness.
- We conduct extensive experiments and ablation studies to evaluate the performance of *Perthept* on four datasets. Furthermore, we build a reasoning-required code datasets for future research by finely filtering the code samples from these well-studied datasets.

2 Robustness Issue of ChatGPT

The decline in ChatGPT's robustness in code comment generation can be described as the following process. When a minor adversarial perturbation σ is introduced into the original code x, the final input code $x + \sigma$ will dramatically degrade model's performance [Zhou *et al.*2022].

A high-quality code summary encompasses two critical aspects: the inclusion of a main purpose and information. However, minor adversarial perturbations often have a great impact on the loss of the main purpose in code summaries. Take the case in Figure 1 as an example, the term "unpack" is the main purpose, and the sentence "the compressed character translation table" is the information. From the perspective of code comprehension during software development, the term "decompress", "unpack", or "unzip" signifies the primary purpose of the code, which are the keywords for precise code comprehension. Therefore, the key to maintaining robustness for ChatGPT in code comment generation lies in not only accurately extracting the main purpose but also preserving sufficient information throughout the process.

3 Perthept

This study proposes a modular prompting framework *Perthept* to address the robustness challenges. The unique contribution of *Perthept* is maintaining robustness for ChatGPT in code comprehension, which is highlighted by successfully reasoning the main purpose of the code. *Perthept* consists of two key prompting paradigms, *Chain-of-Structure* and *Reasoning-Enhancement*. *Chain-of-Structure* preserves robust information from code structures, thus mitigating the potential negative effects resulting from unreliable semantics of code tokens. *Reasoning-Enhancement* paradigm focuses on improving the quality of generated code summaries by iteratively reasoning the main purpose of code while mitigating the inherent unreliability of ChatGPT's responses.

3.1 Chain-of-Structure

The function of *Chain-of-Structure* is two-fold. First, decompose the complex programming language into natural language-based information to facilitate main purpose reasoning. Second, preserving reliable information by shifting the attention of ChatGPT from unreliable semantics of code tokens into reliable code structure information.

Accurately producing *Chain-of-Structure* with necessary information is crucial. The limited context space imposes two primary concerns in *Chain-of-Structure* prompting design: the length of outputs and the level of information detail. The prompting paradigm can be categorized into three different granularities, fine-grain, middle-grain, and coarse-grain, based on the level of detailed information that is incorporated.

Figure 3 shows how different granularities affect the information quality generated by ChatGPT. The fine-grained *Chain-of-Structure* focuses on the smallest units of the code, such as statements, expressions, variables, etc. This level of analysis provides detailed insights and precise understanding. However, it can also increase complexity and make the code analysis more intricate due to its lengthy generated text. The middle-grained *Chain-of-Structure* analyzes the code at a higher level of abstraction, considering functions, classes, modules, or components. This level provides a balance between fine-grain details and overall structure. The coarse-grained *Chain-of-Structure* analyzes the code at a high-level overview, such as architectural patterns, code design, or the overall flow of the program. This level of analysis focuses on understanding the large-scale structure and interactions between major components.

The ideal *Chain-of-Structure* should provide a clear description of code patterns and architectures since they form the foundation of code and represent its primary focus. It has been observed that both fine-grained and middle-grained *Chain-of-Structure* tend to prioritize detailed components such as expressions and variables while overlooking higher-level design patterns and architectural considerations. On the other hand, the coarse-grained *Chain-of-Structure* places more emphasis on these broader aspects. In this study, we adopt the few-shots learning approach and include several coarse-grained examples in the prompting pattern to generate *Chain-of-Structure*. Implementation details are included in the Appendix A.1.1.

Source Code

```
private static char[] func(String p){
    char[] map=new char[0x10000];
    int i=0;
    int j=0;
    while (i <112) {
        int count=p.charAt(i++);
        char value=p.charAt(i++);
        do map[j++]=value;
        while (–count >0);
    }
    return map;
}
```

Fine Grained

Here's the chain of structure for the given code:

1. Initialize an empty character array map with a size of 0x10000.
2. Initialize variables i and j to 0.
3. Start a while loop that continues until i is less than 112.
4. Within the loop, extract the count value from the p string at index i and assign it to the variable count.
5. Increment i to move to the next position in the string.
6. Extract the value from the p string at index i and assign it to the variable value.
7. Increment i to move to the next position in the string.
8. Enter a do-while loop that iterates count number of times.
9. Assign the value to the j-th index of the map array.
10. Increment j to move to the next position in the array.
11. Decrement count by 1.
12. Repeat steps 8-11 until count becomes 0.
13. Continue the outer while loop until i is no longer less than 112.
14. Return the populated map array.

Middle Grained

This code describes the implementation of a function called func, which expects a string p as input and returns a character array map. The function follows a specific structure to populate the map array based on the values and counts provided in the input string. It initializes an empty array, iterates through p, extracting counts and values in pairs, and uses them to populate map by repeating each value a specified number of times. The function ensures that all values and counts in p are processed and the resulting map array is returned as the final output.

Coarse Grained

This code snippet represents a function that takes a string p as input and returns a character array map. It follows a specific structure to populate the map array based on the values and counts provided in the input string.

Figure 3: This figure provides an overview of how different granularity affects the context space and information details. We finally choose to use coarse grain as it not only saves much context space but also provides a high-level overview of the code structure.

3.2 Reasoning-Enhancement

Although *Chain-of-Structure* provides decomposes complex programming language information into simple natural language information, ChatGPT still struggles to effec-

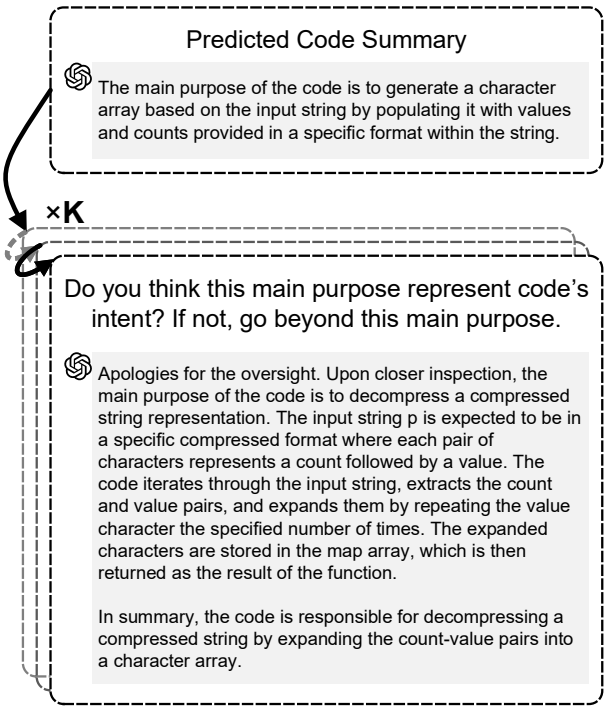


Figure 4: This figure provides an overview of the *Reasoning-Enhancement* process. Through multiple iterations of reasoning enhancement, *Perthept* is able to correct the main purpose from "generate a character array" to "decompressing a compressed string".

tively summarize the main purpose of code from Chain-of-Structure. As depicted in Predicted Code Summary in Figure 4, ChatGPT summarizes the main purpose of code as "generating a character array", which is incorrect and at a low level of comprehension, indicating the importance of *Reasoning-Enhancement* in improving code summary quality.

Reasoning-Enhancement aims at leveraging the ability of ChatGPT to evaluate and refine its predictions to effectively enhance the depth of code comprehension, thus generating insightful and reliable code summaries. It is important to note that even with the same input, the predictions produced by ChatGPT cannot be guaranteed to remain consistent at all times, mainly due to the inherent unreliability of ChatGPT, which means ChatGPT is not guaranteed to catch the code's main purpose all the time. To address this issue, we design the *Reasoning-Enhancement* prompting to be iterative for continuously enhancing the reasoning level. As a result, this approach effectively mitigates the potential negative impact caused by such unreliability.

Reasoning-Enhancement is a self-refinement prompting that prompts ChatGPT to improve its outputs, and it consists of two prompting components: self-evaluation and further-enhancement. For self-evaluation, we use the trigger sentence "Do you think this main purpose represents code's intent?" that enables ChatGPT to evaluate its reasoning results. For further-enhancement, we adopt "if not, go beyond this main purpose for a higher-level abstraction" to ensure that each subsequent prediction during iteration is at a higher level of reasoning. These two prompting components are integrated together and iteratively prompting ChatGPT to perform higher-level reasoning, resulting in the production of reliable code summaries and effectively mitigating the negative impact of the inherent unreliability of ChatGPT's responses. As observed in the final code summary in Figure 4, ChatGPT successfully corrects the main purpose from "generate a character array" to "decompressing a compressed string". More implementation details are included in the Appendix.

4 Experiments

4.1 Setups

Datasets. We evaluate our workflow on four publicly available datasets and one filtered datasets collected on our own, respectively. The statistics of each datasets are shown in Table 2 in Appendix B.1, and the algorithm of reasoning-required code samples collection is included in Appendix B.2.

Adversarial Transformations. We follow [Sontakke *et al.*2022] to use javalang[3] and ast[4] packages to programmatically transform the codes, which includes: (a) Replacing meaningful user-defined function names with more generic function names (e.g., `func`). (b) Replacing meaningful user-defined local variable names with more generic variable names (e.g., `var1, var2`, et al) such that data dependencies are preserved.

Ground Truth Refine. To ensure a fair comparison between NLP and prompting baselines, we refine the ground truth by combining high-level abstraction summaries and brief descriptions. More information is provided in Appendix A.1.2.

Baselines. We consider the following prompting paradigms and NLP models: (a) **Direct Prompt**: a standard zero-shot paradigm that prompts an LLM to directly solve tasks or answer questions. (b) **Chain-of-Thought** [Kojima *et al.*2022]: a prompting paradigm that engages LLMs into "think step by step" reasoning. (c) **SELF-REFINE** [Madaan *et al.*2023]: a prompting paradigm that improves initial outputs from LLMs through iterative feedback and refinement. (d) **CodeBERT** [Feng *et al.*2020]: it is pre-trained on vast bimodal corpora with the masked language model and replaced token detection. (e) **UniXcoder** [Guo *et al.*2022]: it is a pre-trained model that utilizes multi-modal data for better code fragment representation through contrastive learning.

Evaluation Metrics. Common performance metrics such as character-level **Precision** and **F1** score are employed in our experiments. Moreover, a ChatGPT-based scorer is used to measure the semantic **Accuracy** of answers (e.g., "Get" and "Retrieve" are semantically close but cannot be identified by Precision and F1). More information is provided in Appendix A.4.

4.2 Results and Discussions

[3]https://github.com/c2nes/javalang

[4]https://docs.python.org/3/library/ast.html#

Dataset	Paradigm	Acc_O	Pre_O	F1_O	Acc_T	Pre_T	F1_T	Drop_{Acc}	Drop_{Pre}	Drop_{F1}
Python	CodeBERT	—	46.98	39.87	—	18.89	13.33	—	28.09(↓)	26.54(↓)
	UniXcoder	—	56.58	46.62	—	27.73	20.18	—	28.85(↓)	26.44(↓)
	Direct	81.03	46.93	38.73	72.89	40.33	34.13	8.14(↓)	6.60(↓)	4.60(↓)
	CoT	79.69	45.74	40.47	73.77	39.45	35.73	5.92(↓)	5.27(↓)	4.74(↓)
	SELF-REFINE	78.88	44.39	39.58	71.65	38.98	34.22	7.23(↓)	5.41(↓)	5.36(↓)
	Perthept	82.29	47.72	42.59	77.55	45.31	39.26	**4.74**(↓)	**2.41**(↓)	**3.33**(↓)
TL-CodeSum	CodeBERT	—	50.26	41.99	—	42.37	36.48	—	7.89(↓)	5.51(↓)
	UniXcoder	—	57.88	47.42	—	40.89	38.14	—	16.99(↓)	9.28(↓)
	Direct	79.90	57.22	45.18	72.50	53.05	41.63	7.40(↓)	4.17(↓)	3.55(↓)
	CoT	79.55	56.09	45.75	72.99	52.86	42.03	6.56(↓)	3.23(↓)	3.72(↓)
	SELF-REFINE	76.36	54.11	44.76	70.29	49.88	39.85	6.07(↓)	4.23(↓)	4.91(↓)
	Perthept	82.61	55.74	46.00	78.44	53.67	43.80	**3.17**(↓)	**2.07**(↓)	**2.20**(↓)
CSN-Python	CodeBERT	—	50.41	37.61	—	28.34	21.31	—	22.07(↓)	16.3(↓)
	UniXcoder	—	49.40	37.36	—	30.18	24.45	—	19.22(↓)	12.91(↓)
	Direct	69.69	47.66	35.93	64.28	42.99	31.76	5.41(↓)	4.67(↓)	4.17(↓)
	CoT	70.65	47.74	37.71	66.49	44.21	33.28	4.16(↓)	3.53(↓)	4.43(↓)
	SELF-REFINE	67.29	46.88	38.19	61.39	43.36	36.01	5.90(↓)	3.52(↓)	2.18(↓)
	Perthept	72.69	46.67	37.03	68.95	45.74	34.99	**3.74**(↓)	**0.93**(↓)	**2.04**(↓)
CSN-Java	CodeBERT	—	51.16	43.71	—	40.66	36.42	—	10.5(↓)	7.29(↓)
	UniXcoder	—	54.83	43.90	—	42.46	35.60	—	12.37(↓)	8.30(↓)
	Direct	82.33	56.18	47.99	77.17	52.36	43.66	5.16(↓)	3.82(↓)	4.33(↓)
	CoT	85.30	57.05	48.42	79.10	53.08	44.74	6.20(↓)	3.97(↓)	3.68(↓)
	SELF-REFINE	80.08	56.66	47.35	73.18	53.02	43.91	6.90(↓)	3.64(↓)	3.44(↓)
	Perthept	83.22	58.04	48.57	81.30	55.80	46.28	**1.92**(↓)	**2.66**(↓)	**2.29**(↓)
Reasoning-required Ours	Direct	83.04	59.57	48.02	73.10	45.71	36.62	9.94(↓)	17.85(↓)	11.40(↓)
	CoT	80.70	60.51	46.57	74.27	44.54	36.20	6.43(↓)	15.97(↓)	10.37(↓)
	SELF-REFINE	81.87	57.90	47.09	75.38	46.67	36.94	6.49(↓)	11.23(↓)	10.15(↓)
	Perthept	83.04	62.36	47.51	77.78	54.03	38.56	**5.26**(↓)	**8.33**(↓)	**8.95**(↓)

Table 1: Evaluation results. O represents tested on original codes, and T represents tested on transformed codes. *Chain-of-Structure* is provided to each baseline prompting paradigm to ensure a fair comparison because we have used *Chain-of-Structure* to refine the ground truth summary. We omit the Accuracy score for NLP baselines since their scores are significantly low and do not make a substantial difference (e.g., Acc_O is 33.0 while Acc_T is 31.0). The lower score drop and the higher score on each evaluation metric represent a better robustness maintenance performance. Underline: Best performing paradigm. **Bold**: The most robust paradigm.

Perthept excels over all baselines consistently. Table 1 shows the main evaluation results on public datasets based on two popular encoder-decoder models and gpt-3.5-turbo. For the prompting paradigm, we test each code sample twice on each paradigm and retain the samples where there is no significant difference ($< 5\%$) in evaluation scores between two testing results. The drop in Accuracy scores demonstrates the main purpose reasoning performance for each prompting paradigm. A lower Accuracy drop signifies a more robust performance of the prompting paradigm on ChatGPT. The drop in Precision and F1 scores reflects the loss of essential information. A lower drop in Precision and F1 scores indicates the code summary retains a higher amount of information.

Under the same setups, we observe that the prompting paradigms both demonstrate significantly higher levels of robustness compared to NLP baselines. This is due to the less robust training approach for NLP models, as well as the relatively small size of both the models and datasets involved. On the contrary, ChatGPT is less susceptible to benign perturbations and can maintain consistent performance. Among prompting paradigms, we observe the sheer win of *Perthept*. The baseline prompting approaches manage to maintain relatively low drops in Precision and F1 scores compared to NLP baselines due to ChatGPT's proficiency in generation tasks. However, their Accuracy drops are comparatively higher, indicating a higher incidence of incorrect purpose reasoning, particularly when dealing with complex code scenarios that require main purpose reasoning.

It is worth noting that the majority of code samples in public datasets are relatively simple, and intricate main-purpose reasoning is not often required. However, the extensive volume of data can still provide a rough assessment of the performance of each prompting paradigm, even though the differences may not be substantial. In contrast to baseline prompt-

ing paradigms, *Perthept* achieves higher scores for less robust transformed codes while experiencing lower score drops, indicating that it not only maintains a high level of robustness by successfully abstracting the main purpose of complex code samples but also ensures a high quality of code summaries at the same time. During the evaluation on our Reasoning-required datasets, *Perthept* demonstrates superior performance compared to baseline approaches. It achieves high scores while maintaining low score drops, indicating the effectiveness of main purpose reasoning in robustness improvement for code comprehension.

Why Perthept outperforms all prompting baselines in preserving robustness. It is observed that the baseline prompting paradigm lacks clear guidance for reasoning. When utilizing the same baseline approach to prompt ChatGPT multiple times, ChatGPT tends to modify the way it describes the same concept at the same reasoning depth in each response, instead of offering higher-level insights. In contrast, with the guidance of main purpose reasoning, *Perthept* systematically enhances the depth of reasoning to provide deeper insights, which is the key approach to outperforming all other prompting baselines. We conduct a side experiment where we remove the guidance keywords "with code's main purpose" from *Perthept* prompting. We observed a substantial drop in the performance of *Perthept*, and the reasoning level in each iteration remained unchanged. This experiment highlights the significance of guidance keywords in the task of code comment generation.

Perthept is relatively robust upon ChatGPT's unreliability. Given the crucial semantic information provided by function and variable names, ChatGPT does not consistently generate correct code summaries due to its unreliability. The unreliability of ChatGPT can be characterized by the fact that it generates different code summaries for the same input code and prompts on each iteration. This explains the score discrepancy among the prompting paradigms when evaluating Accuracy scores on original codes. Take direct prompt as an example, ChatGPT occasionally generates incorrect code summaries by providing verbose and redundant descriptions instead of accurately extracting the main purpose of the code. This behavior could be attributed to the lack of clear guidance in the prompting paradigm, which results in ChatGPT not consistently prioritizing the main purpose. This issue also affects Chain of Thought and SELF-REFINE prompting paradigms. On the contrary, the design of Perthept is based on purpose-reasoning which effectively mitigates the impact of unreliability by progressively enhancing the code's main purpose at a higher-level abstraction during each iteration. Through observation, ChatGPT maintains the same response when it is unable to provide a higher-level abstraction, rather than producing incorrect code summaries, thus preserving the robustness. A more specific case study is provided in Appendix A.3.

4.3 Ablation Study

Without Reasoning-Enhancement prompting. The *Reasoning-Enhancement* module plays a crucial role in accurately abstracting the main purpose of the code. As depicted by the orange bar in Figure 5a, the removal of the *Reasoning-Enhancement* module results in an Accuracy drop on all benchmarks, indicating that the *Reasoning-Enhancement* module effectively improves the robustness of complex code comprehension for ChatGPT.

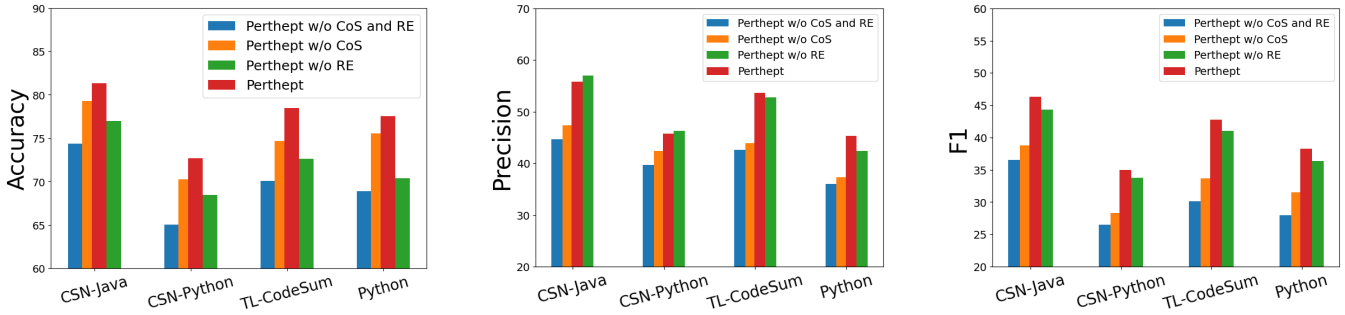
Without Chain-of-Structure prompting. For accuracy scores, although *Chain-of-Structure* may not preserve the higher-level reasoning capabilities, it plays a crucial role in preserving important information, which is vital for generating high-quality code comments. Additionally, the architectural design of *Chain-of-Structure* follows the chain-of-thought prompting paradigm, breaking down complex programming language constructs into the step-by-step analysis. Therefore, this approach inherently incorporates some degree of reasoning, which helps explain why *Perthept* with only *Chain-of-Structure* can achieve higher performance compared to *Perthept* with only the direct summary paradigm of "summarize the code with code's main purpose". Furthermore, based on our observations, if we do not enhance the code's main purpose during the prompting process, there is a risk of generating incorrect summarized main purposes, which does not provide insights into code summary but makes code summary much longer. This can have a detrimental effect on the overall quality of the code summary when testing on a large number of code samples. For precision and F1 scores, the removal of *Chain-of-Structure* leads to a substantial drop in the scores, indicating that the function of *Chain-of-Structure* is more focused on information preservation rather than reasoning.

K value for Reasoning-Enhancement paradigm. Another input of *Perthept* is a user-defined value `K`, representing the number of *Reasoning-Enhancement* paradigm included in *Perthept*. For instance, 0 represents no such paradigm, and 2 illustrates that we repeat the *Reasoning-Enhancement* paradigm once. As depicted by Figure 6, the performance is primarily influenced by the presence or absence of the *Reasoning-Enhancement* paradigm, while the number of repetitions has a limited impact on accuracy improvement. However, to mitigate the unreliability of ChatGPT generations, we finally choose 3 as the K value in this study (i.e., repeat twice).

Additional Observations Upon analyzing the self-feedback generated by ChatGPT, we discover that the suggestions provided by SELF-REFINE prompting rarely focus on providing deeper insights or higher-level reasoning in each self-refine process. Due to the influence of unrelated self-refine suggestions, the refined code summary not only suffers from information loss but also deviates from the main purpose of the code. As a result, the overall generation quality becomes worse.

5 Related Work

Prior to the emergence of extremely large language models (e.g., ChatGPT, GPT4, et al), the most widely adopted approach for code comment generation involves training [Hu *et al.*2018a, Wei *et al.*2019] and fine-tuning [Feng *et al.*2020, Ahmad *et al.*2021, Guo *et al.*2022] specific encoder-decoder



(a) Accuracy on Transformed Codes (b) Precision on Transformed Codes (c) F1 on Transformed Codes

Figure 5: Ablation study. While each prompting paradigm can easily provide correct summaries for simple code samples, the evaluation score can exhibit significant differences across these code samples. We select 3500 samples for each dataset where each prompting paradigm demonstrates similar performance to conduct the ablation study for *Perthept*. We do not set ablation studies on our reasoning-required datasets due to the limited code samples.

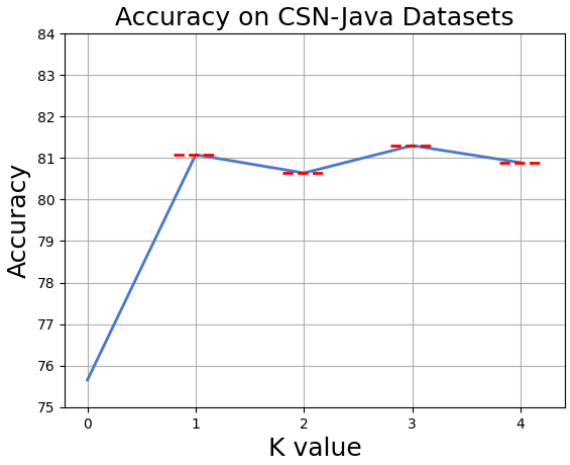


Figure 6: Accuracy scores on different value of K.

models. Among these works, several advancements, including incorporating various attention mechanisms, have demonstrated state-of-the-art performance in generating accurate code comments [Ahmad *et al.*2020,Wu *et al.*2020]. However, these state-of-the-art models both face robustness challenges, as they exhibit poor performance when subjected to even minor perturbations in the input code [Sontakke *et al.*2022]. To mitigate the negative impact of minor perturbations, adversarial training [Goodfellow *et al.*2014] has been widely adopted to enhance model performance [Zhou *et al.*2022, Jha and Reddy2022]. However, due to the limited generalization of the model itself due to small training datasets, these improved models still encounter robustness challenges when presented with new code patterns.

With the rapid development of LLMs, the code comment generation approach is evolving to prompt LLMs. ChatGPT is a recent chatbot service released by OpenAI [Wang *et al.*2023] and has shown great potential in generating coherent text, which can be leveraged for assisting code comprehension during development. However, ChatGPT still faces robustness challenges in consistently producing high-quality code summaries because it relies heavily on the unreliable semantics of code tokens, which can be easily altered by introducing even minor perturbations, resulting in generating useless and lengthy descriptions of code.

As fine-tuning ChatGPT is costly, prompting design emerges as a practical solution to address the robustness challenges. A thoughtfully designed prompt can significantly impact ChatGPT's predictions. Recent studies have shown that the utilization of reasoning prompts can significantly enhance the accuracy of predictions. Recent popular prompting approaches are optimized to allow a single large language model to better perform a variety of tasks [Li and Liang2021, Lester *et al.*2021, Reif *et al.*2021]. Among these works, [Wei *et al.*2022] proposes to apply chain-of-thought prompts to improve the ability of large language models to perform complex reasoning to solve complex tasks. Nevertheless, neither of these works demonstrates a direct impact on addressing the robustness issues for LLMs to improve code comprehension.

In contrast to previous studies, our research specifically aims to activate ChatGPT's purpose-reasoning capability to enhance code comprehension while mitigating the adverse impact caused by robustness perturbations.

6 Conclusion

We present *Perthept*, a modular prompting framework to solve robustness issues efficiently in code comprehension for ChatGPT by leveraging main-purpose reasoning guidance and iterative reasoning enhancement. With clear reasoning guidance and iterative higher-level reasoning enhancement, *Perthept* effectively maintains high levels of robustness in complex code comprehension tasks. Comprehensive experiments reveal superior performance of *Perthept* in achieving a high level of robustness while simultaneously ensuring high-quality code comment generation. A side study also shows that *Perthept* has relatively robust performance under the unreliability of ChatGPT generations. Future improvements beyond *Perthept* involve prompting optimization (e.g., token reduction), robustness improvement for dynamic code comprehension, modular LLM fine-tuning, and augmentation with tools.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work is partially funded by NSF 2050007 and a Google gift.

References

[Ahmad *et al.*, 2020] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[Ahmad *et al.*, 2021] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[de Souza *et al.*, 2005] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. 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*, pages 68–75, 2005.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[Goodfellow *et al.*, 2014] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[Guo *et al.*, 2022] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[He, 2019] Hao He. Understanding source code comments at large-scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1217–1219, 2019.

[Hu *et al.*, 2018a] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*, pages 200–210, 2018.

[Hu *et al.*, 2018b] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. 2018.

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

[Jha and Reddy, 2022] Akshita Jha and Chandan K Reddy. Codeattack: Code-based adversarial attacks for pre-trained programming language models. *arXiv preprint arXiv:2206.00052*, 2022.

[Kojima *et al.*, 2022] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

[Lester *et al.*, 2021] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[Li and Liang, 2021] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[Madaan *et al.*, 2023] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

[Panichella *et al.*, 2016] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th international conference on software engineering*, pages 547–558, 2016.

[Reif *et al.*, 2021] Emily Reif, Daphne Ippolito, Ann Yuan, Andy Coenen, Chris Callison-Burch, and Jason Wei. A recipe for arbitrary text style transfer with large language models. *arXiv preprint arXiv:2109.03910*, 2021.

[Sontakke *et al.*, 2022] Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. Code summarization: Do transformers really understand code? In *Deep Learning for Code Workshop*, 2022.

[Sridhara *et al.*, 2010] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[Wan *et al.*, 2018] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 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, 2018.

[Wang *et al.*, 2023] Jindong Wang, Xixu Hu, Wenxin Hou, Hao Chen, Runkai Zheng, Yidong Wang, Linyi Yang, Haojun Huang, Wei Ye, Xiubo Geng, et al. On the robustness of chatgpt: An adversarial and out-of-distribution perspective. *arXiv preprint arXiv:2302.12095*, 2023.

[Wei *et al.*, 2019] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32, 2019.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[Wu *et al.*, 2020] Hongqiu Wu, Hai Zhao, and Min Zhang. Code summarization with structure-induced transformer. *arXiv preprint arXiv:2012.14710*, 2020.

[Zhou *et al.*, 2022] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald Gall. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–30, 2022.