CoRAC: Integrating Selective API Document Retrieval with Question Semantic Intent for Code Question Answering

YunSeok Choi, CheolWon Na, Jee-Hyong Lee[†]
College of Computing and Informatics
Sungkyunkwan University, South Korea
{ys.choi, ncw0034, john}@skku.edu

Abstract

Automatic code question answering aims to generate precise answers to questions about code by analyzing code snippets. To provide an appropriate answer, it is necessary to accurately understand the relevant part of the code and correctly interpret the intent of the question. However, in real-world scenarios, the questioner often provides only a portion of the code along with the question, making it challenging to find an answer. The responder should be capable of providing a suitable answer using such limited information. We propose a knowledge-based framework, CoRAC, an automatic code question responder that enhances understanding through selective API document retrieval and question semantic intent clustering. We evaluate our method on three real-world benchmark datasets and demonstrate its effectiveness through various experiments. We also show that our method can generate high-quality answers compared to large language models, such as ChatGPT.

1 Introduction

Online communities such as Stack Overflow, Reddit, and GitHub have become essential resources for developers seeking solutions to coding challenges. Developers frequently rely on these platforms to overcome obstacles, deepen their understanding of programming concepts, and enhance their productivity. These communities serve as knowledge hubs for a wide audience, ranging from professional software developers to students studying programming, and they have accumulated vast amounts of information. The growing volume of question-and-answer interactions on these platforms has led to a need for automated methods to efficiently provide accurate and relevant answers to code-related queries.

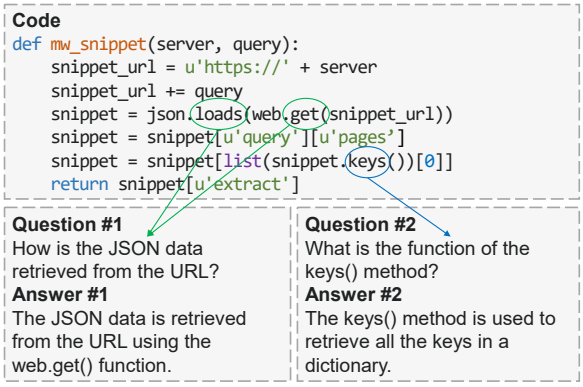

Figure 1: An illustration of code question answering. The knowledge of necessary API functions varies depending on the specific question and the intended purpose behind each question also differs.

Recently, many pre-trained language models (PLMs) for programming and natural language have been developed, such as CodeBERT (Feng et al., 2020), PLBART (Ahmad et al., 2021), UnixCoder (Guo et al., 2022), CodeT5 (Wang et al., 2021), and CodeGen (Nijkamp et al., 2023). These models have achieved remarkable success across various code understanding tasks, including defect detection (Zhou et al., 2019), clone detection (Svajlenko et al., 2014; Mou et al., 2016), code translation (Lu et al., 2021), code summarization (Husain et al., 2019), and code generation (Iyer et al., 2018). This success is due to the utilization of large-scale corpora of code and text. These PLMs learn not only from code (uni-modal data) but also code-comment pairs (bi-modal data) in the pre-training stage. This allows the models to learn both the semantic information of code and the relationships between code and natural language. These models have shown significant performance on various tasks after fine-tuning. Liu and Wan (2021) also showed the possibility of PLMs for code question answering.

However, these PLMs were fine-tuned by sim-

[†] Corresponding author.

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 12620–12635
April 29 - May 4, 2025

ply providing a pair of code and question as input and the answer as output, which is the typical approach for tasks such as code translation and code summarization. Unlike these tasks, code question answering requires a deeper understanding of the code's meaning and context, along with the ability to make inferences. There are three main aspects to the challenging issues of code question answering. First, answering questions about code often requires capabilities for inference and reasoning. This requires a higher level of understanding and reasoning than just generating or translating code. Second, it needs to recognize which API functions are utilized in the given code and understand the purpose and usage of those API functions. This is crucial for understanding the usage and functionality of the API function and providing accurate responses to code questions. Third, code questions can encompass a variety of problem types. For instance, they might involve tasks such as debugging code errors, solving algorithmic problems, reasoning, or providing explanations. The model needs to comprehend these various problem types.

To address these challenging aspects of code question answering, we utilize specific domain knowledge used within the code and a precise understanding of the question' intent to generate answers. For example, as shown in Figure 1, responders should have knowledge of the API functions used in the code, such as `loads()`, `get()`, and `keys()`, to understand the code and infer the answer. If they lack knowledge of it, API documentation can provide information on how to use functions and their parameters.

The key point is to identify the API functions relevant to the question being asked. Question #1 involves understanding API function calls related to JSON, while Question #2 specifically requires knowledge of the `keys()` function. Therefore, it is important to identify which API function documentation is helpful in answering the questions.

Furthermore, to provide more accurate and relevant answers, responders should understand the intent of the questions. As shown in Figure 1, Question #1 asks about the process of retrieving JSON data from a URL using the given code, so the answer should provide information about `Usage of APIs`. In contrast, Question #2 inquires about the purpose and functionality of the given code, so the answer should provide `Purpose` or `Functionality` information. By understanding the underlying purpose of the question, responders can generate a more tailored and relevant answer that directly addresses the asker's concerns.

However, in real-world scenarios, questioners often provide only code and questions without explicitly indicating their question intent. Misinterpreting the intent can result in answers that fail to meet the questioner's expectations. For example, if a question seeks the "purpose" of a code block but the response focuses on its "usage", the answer may mislead the questioner even when utilizing specific domain knowledge such as API documentation. Thus, accurately understanding the intent of questions is crucial in generating accurate and relevant answers.

In this paper, we propose a knowledge-based framework, **CoRAC**, an automatic **Co**de question **R**esponder by utilizing **A**PI documentation as external domain knowledge and **C**lustered question intent instruction. We propose selective API document retrieval that extracts helpful API function descriptions highly relevant to the question through an external PLM. Also, we design question intent instruction that represents the semantic intent of the question about the code. By training question semantic intent instruction with the cluster prompt template, we can indirectly inform the type of question and guide the model to improve the quality of answers to code-related questions. To demonstrate the effectiveness of our model, we conduct experiments on three real-world datasets: two Github datasets (Java and Python) and one introductory programming course dataset (Python). We show that our model can generate high-quality answers compared to recent large language models (LLMs).

2 Related Work

Automatic question answering has been consistently studied in the field of natural language processing (Yang et al., 2015; Rajpurkar et al., 2016, 2018; Yang et al., 2018). Recently, attention has shifted toward automatic code question answering in program language processing. Xu et al. (2017) introduced a system that generates answers to technical questions from StackOverflow. Similarly, Bansal et al. (2021) designed an answering system for questions about subroutines using an RNN encoder-decoder network (Cho et al., 2014). However, questions in these datasets tend to be simple, where answers can be easily obtained without requiring complex inference or reasoning about the code.

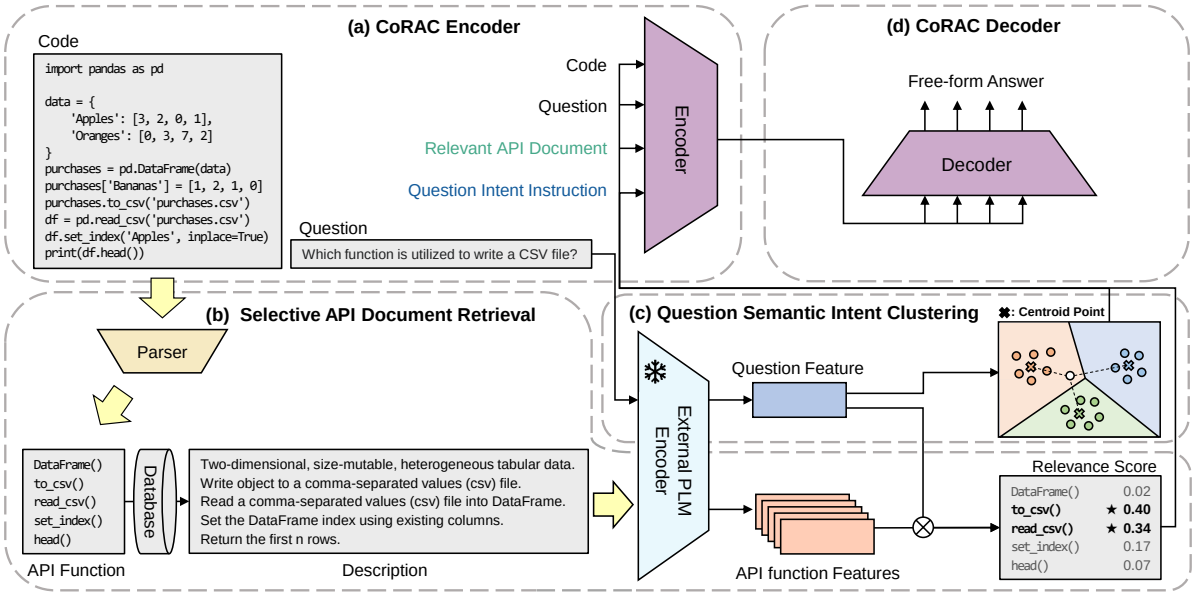

Figure 2: Overview of CoRAC framework. (a) For code question answering, not only code and question but also important API documents selected by our method and instruction containing the semantic intent of the question are given as inputs for training the end-to-end model (Sec. 3.1). (b) For selecting helpful API documents, we parse the code to extract all API functions and match them with the corresponding database for each programming language to obtain the description of the respective API function. We then select the important API documentation relevant to the question based on semantic features utilizing an external PLM (Sec. 3.2). (c) To grasp the intent of the question, we create a question semantic intent instruction from clustering-based questions based on the semantic features (Sec. 3.3). (d) With the selected important API function's documents and question intent instruction, the decoder is trained to generate free-form answers (Sec. 3.4).

Liu and Wan (2021) constructed a more complex code question answering dataset, which consists of free-form answers requiring a deep understanding of both the code and the associated question to provide comprehensive answers. They were the first to apply this dataset to the PLMs such as CodeBERT (Feng et al., 2020) for the task. Additionally, Lee et al. (2022) introduced a more real-world dataset, CS1QA, that focuses on the introductory programming education domain.

Recently, the rise of LLMs in programming and natural language processing, such as Llama3.1 (Dubey et al., 2024), Code Llama (Roziere et al., 2023), Qwen2.5-Coder (Hui et al., 2024) has led to notable improvements in various code intelligence tasks. However, when it comes to code question answering, these LLMs often produce responses that are overly detailed and focused on the code itself, which makes it difficult to align with the user's specific intent or request. GPT-4 (OpenAI et al., 2024) also shows impressive abilities across various domains due to its extensive training with vast data, resources, and instructions. Yet, when given only fragments of code and questions, GPT-4 also struggles to provide accurate answers and it also requires well-structured prompts for proper responses.

3 Proposed Method

In this section, we introduce a knowledge-based framework, CoRAC, to leverage helpful API documentation related to the question and to utilize the semantic intent of the question as shown in Figure 2.

3.1 CoRAC Encoder

Given a set of source code C and a natural language question Q, the goal of the code question answering task is to generate a free-form natural language answer A. This task requires understanding the entire code to generate an appropriate answer. However, when questioners ask about code, they provide only a portion of it, making it challenging for the respondent to give an accurate answer based on the limited code. To gain a deeper understanding of the code in question, we incorporate external knowledge by leveraging question-related essential API documents and question intent instructions for guiding the proper answer. Our CoRAC encodes

all inputs into features as follows:

$$X = [C; S; Q; I] \quad (1)$$
$$\mathrm{F}^X = \mathrm{Encoder}_{\mathrm{CT}}(X)$$

where C, S, Q, and I denote the source code, the retrieved API documentation, the question, and the question semantic intent instruction, respectively, and $\mathrm{Encoder}_{\mathrm{CT}}$ is the Code-Text PLM encoder. We concatenate the inputs as $X = [C; S; Q; I]$.

In the following subsection, we describe the methods for selecting the important API description (Figure 2 (b)) and for obtaining the question semantic intent instruction (Figure 2 (c)).

3.2 Selective API Document Retrieval

To extract API functions used in code for code question answering, we first parse the source code as an Abstract Syntax Tree (AST) parser. By traversing the AST nodes, we check whether a node represents an API function call, and we extract the name of the API function, such as `to_csv()`. We obtain a list of all API functions used in the code, denoted as $F = \{f_1, f_2, ..., f_l\}$. Next, we use a database of API-document pairs for each programming language and leverage Elastic Search[1] with the standard BM25 (Robertson et al., 2009) to retrieve the document of each API function. Then, the description for all API functions, denoted as $D = \{d_1, d_2, ..., d_l\}$ is obtained and each API function is matched with its corresponding document. We then build API-description pairs, denoted as $FD = \{fd_1, fd_2, ..., fd_l\}$ where $fd_i = [f_i; d_i]$.

However, descriptions of all the functions used in the code do not necessarily help in generating answers to questions. There may be unnecessary API functions that are not relevant to the given question, which can lead to confusion in code question answering. Therefore, we need to consider only the API functions that are relevant to the given question. To select the descriptions of API functions that are helpful in answering the question, we compute a confidence score for all API documentation used in the code based on how relevant they are to the question. To do this, we use an external PLM to obtain semantic features of the question and the descriptions of API function.

$$\mathrm{F}^Q = \mathrm{Encoder}_{\mathrm{Ext}}(Q) \quad (2)$$

$$\mathrm{F}^{fd_i} = \mathrm{Encoder}_{\mathrm{Ext}}(fd_i) \quad (3)$$

[1]https://github.com/elastic/elasticsearch

where the number of F^{fd} is l and $\mathrm{Encoder}_{\mathrm{Ext}}$ is external PLM Encoder. Next, we compute their cosine similarity between the features of the question and all API-descriptions used in the code as:

$$s_i = \mathrm{similarity}(\mathrm{F}^Q, \mathrm{F}^{fd_i}) \quad (4)$$

Finally, we calculate the relevance score of API documents for a given question based on a similarity score.

$$r_i = \frac{e^{s_i}}{\sum_{j=1}^{L} e^{s_j}} \quad (5)$$

Based on the obtained relevance scores, we select the top-n important API function documents S as the input for the CoRAC encoder.

3.3 Question Semantic Intent Clustering

Understanding the intent of the question is crucial in generating accurate answers. It not only helps responders provide more relevant and comprehensive responses but also guides the answer's formulation. Different problem types of questions require different kinds of answers, and recognizing the intent can help generate the appropriate answer.

However, in the real world, questioners typically do not provide specific intent-related information about their questions but rather provide only code and questions. As a result, responders need to infer the intent of the questions solely from the question sentences.

If questions have similar intents and problem types, they should produce similar answers. In situations where intent is not provided for such questions, i.e., in unlabelled scenarios, we propose an effective method to learn question semantic intent.

If we have sufficient data for code-related questions, we first obtain the semantic features of all questions based on an external PLM. Then, we apply the K-means clustering method to group all questions in the training dataset into K clusters.

$$V = \sum_{i=1}^{N} \sum_{j=1}^{K} ||\mathrm{F}^{Q_i} - u_j||^2 \quad (6)$$

where N is the total number of questions for clustering and K is the number of clusters. We obtain the centroid points u of group j minimizing V using Euclidean distance. This can group questions with similar semantic intent together. K centroid ids mean representative points for K semantic intents.

Then, when a question is given in the training or inference stage, we extract the feature of the question using the external PLM and identify the cluster id closest to the centroid points u based on the Euclidean distance.

$$k = \underset{j=1,\ldots,K}{\operatorname{argmin}} ||\mathrm{F}^{Q} - u_j||^2 \quad (7)$$

Then, we use "Question intent is [cluster id]" as the instruction I for the CoRAC encoder.

3.4 CoRAC Decoder

We utilize the PLM decoder for program and language to generate free-form natural language answers. The decoder of our CoRAC takes the feature F^{X} as input, finally generating the answer.

$$y = \mathrm{Decoder}_{\mathrm{CT}}(\mathrm{F}^{C}) \quad (8)$$

where F^{C} is a vector, y denotes the predicted answer, and $\mathrm{Decoder}_{\mathrm{CT}}$ represents the Code-Text PLM decoder model. The decoder generates words in an autoregressive manner until the </s> token is produced.

4 Experiment Setup

4.1 Datasets

We conduct experiments on two public datasets, CodeQA Java and Python (Liu and Wan, 2021), and a more real-life dataset, CS1QA Python dataset (Lee et al., 2022), which is collected from an introductory programming course. Table 6 in Appendix A.1 provides the detailed statistics of the datasets.

Database of API function documentation are obtained from Devdocs[2] API documentation for Python language and Oracles[3] API documentation for Java language. We pre-processed API function documentation and stored about 33,000 pairs for Python language and about 51,000 pairs for Java language.

4.2 Evaluation Metrics

We use three evaluation metrics, smoothed BLEU-4 (Lin and Och, 2004), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004) to measure the quality of the generated answers. The ground truth consists of the answers provided in each dataset, including CodeQA and CS1QA, both of which are hand-curated to ensure quality, accuracy, and relevance. Detailed descriptions of the evaluation metrics are reported in Appendix A.2

[2]https://devdocs.io/

[3]https://docs.oracle.com

4.3 Baselines

We compare our proposed model with two categories: (1) Pre-trained Language Models for code intelligence (Feng et al., 2020; Ahmad et al., 2021; Wang et al., 2021; Guo et al., 2022; Nijkamp et al., 2023), and (2) Large Language Models (Roziere et al., 2023; Hui et al., 2024; Dubey et al., 2024; OpenAI et al., 2024).

- **CodeBERT** (Feng et al., 2020), an encoder-only model based on RoBERTa (Liu et al., 2019), designed for code representation and understanding tasks.
- **PLBART** (Ahmad et al., 2021), an encoder-decoder model to support code generation tasks using BART (Lewis et al., 2020).
- **CodeT5** (Wang et al., 2021), a pre-trained encoder-decoder model based on T5 (Raffel et al., 2010), to facilitate generation tasks for source code.
- **UnixCoder** (Guo et al., 2022), a unified cross-modal pre-trained language model for programming language.
- **CodeGen** (Nijkamp et al., 2023), a pre-trained decoder model for programming language. from 350M up to 16B parameters. They train a family of large language models from 350M up to 16.1B parameters.
- **Code Llama** (Roziere et al., 2023), an advanced LLM, specifically designed to assist with code generation and understanding tasks. We use `CodeLlama-7b-Instruct-hf` model.
- **Llama3.1** (Dubey et al., 2024), a new state-of-the-art model of the LLaMA language model series. This is trained on trillions of tokens. We use the `Llama3.1-8B-Instruct` model.
- **Qwen2.5-Corder** (Hui et al., 2024), a specialized variant of the Qwen language model series, optimized for programming tasks such as code generation. We use the `Qwen2.5-Coder-7B-Instruct` model.
- **GPT-4** (OpenAI et al., 2024), a state-of-the-art LLM developed by OpenAI, designed to handle complex language understanding and generation tasks. We utilized the `GPT-4-o-mini` on 4-shot in-context learning.

Models	#Param	CodeQA (Java)			CodeQA (Python)			CS1QA		
		BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
In-context Learning										
Code Llama (4-shot)	7B	2.45	0.81	2.61	3.54	1.57	4.07	1.87	5.87	5.22
Llama3.1 (4-shot)	8B	2.20	0.90	2.33	3.07	1.77	3.49	1.25	4.01	4.81
Qwen2.5-Corder (4-shot)	7B	2.60	0.99	2.88	3.38	1.46	3.78	1.80	5.62	4.35
GPT-4 (4-shot)	-	14.97	8.58	21.11	18.37	9.41	23.19	4.91	9.30	12.62
Fine-tuning										
CodeBERT	172M	23.93	9.90	27.86	25.37	11.92	29.89	5.51	5.08	12.68
PLBART	139M	25.04	12.04	30.35	28.72	15.70	34.36	7.26	9.59	16.30
UnixCoder	126M	24.11	10.24	26.57	28.63	14.43	31.42	6.06	7.43	14.27
CodeT5	220M	26.59	12.43	32.70	30.18	16.74	36.35	7.31	9.49	16.49
CodeGen	350M	26.69	12.36	32.28	30.03	16.10	36.13	6.71	9.11	14.87
CoRAC (ours)	220M	**28.05**	**13.51**	**33.57**	**32.14**	**17.72**	**38.42**	**7.52**	**9.77**	**18.07**

Table 1: Comparison of our proposed method with the baseline models on the three benchmark datasets. We conducted statistical testing using paired-sample z-tests to confirm the statistical significance of our results. CoRAC shows a statistically significant performance improvement over baselines with $p < 0.01$.

4.4 Implementation Details

We implemented our proposed method and baselines based on the Hugging Face Transformer models[4] (Wolf et al., 2020) on 4 Nvidia 3090ti GPUs (24GB). We selected the CodeT5-base model as the PLMs for program and language. Our models were trained using the Adam optimizer (Loshchilov and Hutter, 2019) with a learning rate of 5e-5 and a linear learning rate scheduler. We set the input code and question length to 300, the target answer length to 32, and the batch size to 32 for all datasets. The maximum training epoch is 20, with early stopping applied if the performance does not improve for 3 epochs. The models generate answers using the beam search with a beam size of 10. For all experiments, including all baselines, we report the mean value of three folds.

For our external PLM, we selected paraphrase-MiniLM, which has only 22 million parameters significantly fewer than other PLMs and LLMs. This is one of the SentenceBERTs to map sentences and paragraphs to a 384-dimensional dense vector space and demonstrates strong performance in tasks such as clustering and semantic search. Based on initial experiments, we set the number of API documents to 3 and the maximum length of API function descriptions to 100. Additionally, we set the number of clusters K to 10 for the cluster-based question semantic intent. More details are discussed in subsection 5.5

[4]https://github.com/huggingface/transformers

5 Experiment Result

5.1 Overall Result

Table 1 shows the comparison between our CoRAC and recent state-of-the-art on three benchmark datasets for code question answering. Among the fine-tuning methods, the CodeT5 model has the best performance in the PLMs for program and language. PLBART and CodeT5 models, which are encoder-decoder models, show better performance than CodeBERT and UnixCoder, which are only encoder models paired with a randomly initialized Transformer decoder. CodeGen-350M, which is only a decoder model, has compatible performance with CodeT5. Our CoRAC shows a significant improvement on the three benchmark datasets. Rather than simply using code and questions as input (CodeT5), incorporating relevant API function documents and question semantic intent instruction results in a much better performance. This demonstrates that API function documents relevant to the question help the model gain a deeper understanding of the code and the question semantic intent instruction aids in generating appropriate responses.

Large language models in a four-shot setting showed poor performance. Code Llama, which was trained mainly for code generation during pre-training, struggled with the natural language answer generation task. While Llama is known to perform adequately in in-context learning for cloze-test question answering, it showed poor performance in our study due to the more challenging requirement of understanding code and generating free-form answers. Although GPT4 showed good

Models	BLEU	METEOR	ROUGE-L
CodeQA (Java)			
CodeT5	26.59	12.43	32.70
+ *SDR*	27.32 (+2.7%)	13.29 (+6.9%)	33.24 (+1.7%)
+ *QSI*	27.91 (+5.0%)	12.63 (+1.6%)	33.20 (+1.5%)
CoRAC	**28.05 (+5.5%)**	**13.51 (+8.7%)**	**33.57 (+2.7%)**
CodeQA (Python)			
CodeT5	30.18	16.74	36.35
+ *SDR*	31.43 (+4.1%)	17.47 (+4.4%)	37.98 (+4.5%)
+ *QSI*	31.59 (+4.7%)	17.36 (+3.7%)	37.83 (+4.1%)
CoRAC	**32.14 (+6.5%)**	**17.72 (+5.9%)**	**38.42 (+5.7%)**
CS1QA			
CodeT5	7.31	9.49	16.49
+ *SDR*	7.42 (+1.5%)	9.37 (-1.3%)	17.49 (+6.1%)
+ *QSI*	7.34 (+0.4%)	9.49 (+0.0%)	17.53 (+6.3%)
CoRAC	**7.52 (+2.8%)**	**9.77 (+3.0%)**	**18.07 (+9.6%)**

Table 2: Ablation Study on two modules of our CoRAC: Selective API Document Retrieval (*SDR*) and Question Semantic Intent Clustering (*QSI*).

performance due to its extensive training on vast datasets, it incurs a significant cost and time. Our CoRAC demonstrates comparable results without incurring such expenses.

5.2 Ablation Study

We conduct an ablation experiment that focuses on two modules: Selective API Document Retrieval (*SDR*) and Question Semantic Intent Clustering (*QSI*). CoRAC (*only SDR*), CoRAC (*only QSI*), and CoRAC (*SDR* + *QSI*) correspond to our models that use only the selective API document retrieval, only the question semantic intent clustering, and both modules, respectively. In Table 2, we observed a significant improvement in code question answering performance when utilizing just the first module, CoRAC (*only SDR*). Instead of merely using code and questions as input, integrating API function documents with the code markedly enhances performance. This demonstrates that our method effectively selects important API documents to address questions relevant to the code. Moreover, CoRAC (*only QIC*) employs the question semantic intent instruction within the CodeT5 model, proving particularly effective, especially in terms of BLEU scores. By training the cluster-based question semantic intent, it helps generate more appropriate responses. Finally, our CoRAC, which uses both relevant API document retrieval and question semantic intent clustering, shows a significant improvement in all evaluation metrics compared to CodeT5.

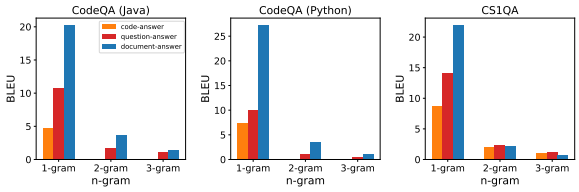

Figure 3: BLEU score of n-gram overlap recall on code-answer (orange), question-answer (red), and document-answer (blue).

PLMs	#Param	BLEU	METEOR	ROUGE-L
CodeQA (Java))				
SBERT	22M	**28.05**	**13.51**	33.57
CodeT5	220M	26.49	12.37	32.58
Llama	7B	27.98	13.47	**33.59**
CodeQA (Python)				
SBERT	22M	32.14	**17.72**	**38.42**
CodeT5	220M	30.09	16.34	36.12
Llama	7B	**32.16**	17.65	38.36
CS1QA				
SBERT	22M	7.52	9.77	18.07
CodeT5	220M	6.97	8.84	15.37
Llama	8B	**7.61**	**9.81**	**18.13**

Table 3: Investigation of external PLMs for semantic feature extraction.

5.3 Effectiveness of API Documents

In Figure 3, we present n-gram overlap recall between code-answer, question-answer, and API document-answer to examine whether the API function documents used in answering the questions are actually helpful. The result shows that the overlaps between code-answer and question-answer are very low in all datasets. There are almost no shared words between code and answer, and there is only a slight overlap between question and answer. However, in the case of API document and answer, we observe that there are many overlapped words, which means that words in API documents but not in code or question help in generating relevant and accurate answers.

5.4 Investigation of External PLMs

We analyzed the results of our investigation into various external PLMs for extracting the semantic features of questions and API documentation, as shown in Table 3. The parameter sizes for SBERT, CodeT5, and Llama are 22M, 220M, and 8B, respectively. CodeT5, a PLM for code and language, is trained solely on code and its corresponding comments rather than a variety of textual contexts. As

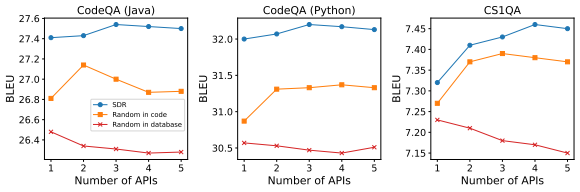

Figure 4: Results for varying numbers of relevant API documents by our selective API document retrieval.

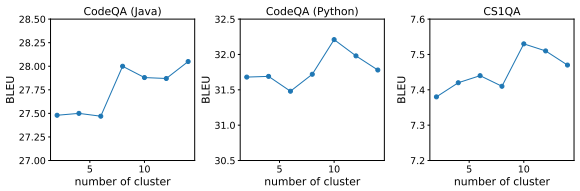

Figure 5: Results for varying numbers of clusters in our question semantic intent clustering.

a result, when obtaining the semantic feature of natural language questions and API documentation, it performed poorly performance in clustering and semantic search tasks. This shows the importance of selecting PLMs appropriate for input relevance rather than relying solely on a single model like CodeT5. While Llama showed performance comparable to SBERT, considering that SBERT's parameter size is over 300 times smaller, it demonstrates that our method effectively performs clustering and semantic search on questions and API documentation even with significantly fewer parameters.

5.5 Exploration of Various Parameters

Number of API Documents Figure 4 presents the results with different numbers of important API documents by selective API document retrieval. We selected API documents based on three criteria: those chosen by our proposed method (blue), those randomly used in the code (orange), and those randomly selected from the database (red), ranging from 1 to 5 API documents. As the number of unrelated API documents (red) increased, there was a noticeable decrease in performance. This can hinder the understanding of the code. When randomly providing documentation of the API functions used in the code (orange), performance improved as the count increased. However, since the length of the input API documentation is limited to 100, the impact was not significantly beneficial. Our method, which selects relevant API documents for generating answers to code-related questions, showed superior performance even with fewer API documents.

Question Intent	BLEU	METEOR	ROUGE-L
CodeQA (Java)			
None	26.59	12.43	32.70
Type	27.14	12.59	33.15
Semantic	**27.91**	**12.63**	**33.20**
CodeQA (Python)			
None	30.18	16.74	36.35
Type	31.28	17.19	37.47
Semantic	**31.59**	**17.36**	**37.83**
CS1QA			
None	7.31	**9.49**	16.49
Type	7.27	9.48	16.74
Semantic	**7.34**	**9.49**	**17.53**

Table 4: Comparison of different question intents approaches.

Number of Clusters In Figure 5, we present the results for different numbers of clusters in our question semantic intent clustering. Grouping the questions into fewer, broader clusters was not as effective as clustering them into more specific categories, such as those with 10 to 14 clusters. This indicates that, due to the high diversity of question types, it is beneficial to group them into a larger number of clusters.

5.6 Analysis of Question Semantic Intent

We compared the effectiveness of the clustered question semantic intent with cases where the actual intent, such as "How", "What", or "Purpose" was provided. Table 4 presents the result across three different conditions: (1) without providing any intent information for the question (*None*), (2) providing the actual intent label of the question (*Type*), and (3) using question semantic intent obtained by our method (*Semantic*). The result shows that providing question intent (*Type*) and (*Semantic*) leads to better performance compared to the CodeT5 model without question intent (*None*). In particular, the question semantic intent (*Semantic*) demonstrates a significant performance improvement compared to using the actual intent label (*Type*). Notably, in the CS1QA dataset, which consists of more complex questions, there is a substantial improvement in ROUGE-L scores. This proves that rather than providing categorized intent from the questioner, the clustered question semantic intent based on semantic features for similar questions provides a more accurate understanding of the question's intent, and generates more accurate answers.

Cluster 1
For what purpose will it be broken if any of the specified rows contains a new-line character on that character?
Does the code dump the parse stack for debugging purposes?
What does the code dump for debugging purposes?
For what purpose do rotation change?
Cluster 2
Does the code convert a number into a bit string with separators between each group of 8?
Does the code convert an unsigned 32-bit integer to a string?
Does the matrix need to be translated after rotating?
Does the code convert it into string?
Cluster 3
What does this method return?
Does the code return the given charset or the default charset if the given charset is null?
What does the code return if the given charset is null?
What does the code add to the output suffix?
Cluster 4
What does the code send to ourselves to update the execution stage?
How does execution cease?
What is performed inside a separate thread of execution?
How can the operation work?

Table 5: Clustering results of question semantic intent in the CodeQA (Java) dataset.

5.7 Qualitative Results of Clustering

Table 5 presents how effectively questions are semantically clustered using our Question Semantic Intent Clustering method. We extracted sample questions closest to each cluster centroid. In the first cluster, questions related to the "purpose" of the code were grouped together. The second cluster consists of questions about code transformation, while the third cluster includes questions regarding the values returned by the code. Examining each example reveals that the questions within each cluster share similar intents and are well-clustered. Therefore, our proposed method effectively clusters questions based on their semantic intent, demonstrating its usefulness in guiding answer generation.

6 Conclusion

We proposed a knowledge-based framework that leveraged helpful API function documents (Selective API Document Retrieval) and instruction of question semantic intent (Question Semantic Intent Clustering) to generate more accurate and relevant answers to code questions. We demonstrated the effectiveness of our methods and highlighted the need for more efficient ways to obtain accurate and relevant answers to code-related inquiries.

In this paper, we focus on generating natural language answers that include concise function calls rather than full function-level code. As future work, we could explore generating answers that integrate textual explanations with full function-level code snippets.

7 Limitation

GPT-4 demonstrates good performance not only in many natural language tasks but also in code intelligence tasks. However, there are issues with human evaluation of GPT4 for the dataset discussed in this paper. The average length of ground truth answers is between 4 to 5 words as shown in Table 1. GPT4 tends to generate lengthy responses even in few-shot prompting. This characteristic makes it easy to detect which method generated an answer, leading to the conclusion that a comparison might not be meaningful. Instead, we presented the qualitative examples generated by GPT4.

For open LLMs like CodeGen and Llama, neither model was trained with PEFTs like LoRA in our GPU setting. CodeGen, as an LLM for code generation, was not sufficiently trained during the pre-training phase to generate natural language in the decoder, which is why it performs poorly on tasks like Code-to-NL. Llama, as an LLM for natural language, was not primarily trained for code data. We chose not to use all LLMs as baselines for our study, as those designed for code generation performed poorly on Code-to-NL tasks. We believe solving this issue is a challenge for the field of code intelligence.

Acknowledgments

This work supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (RS-2019-II190421, AI Graduate School Support Program(Sungkyunkwan University), 10%) (No.1711195788, Development of Flexible SW/HW Conjunctive Solution for on-edge self-supervised learning, 45%) (IITP-2025-RS-2024-00437633, ITRC(Information Technology Research Center), 45%)

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings*

of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2655–2668, Online. Association for Computational Linguistics.

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

A. Bansal, Z. Eberhart, L. Wu, and C. McMillan. 2021. A neural question answering system for basic questions about subroutines. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 60–71, Los Alamitos, CA, USA. IEEE Computer Society.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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 *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *ArXiv preprint*, abs/2009.08366.

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

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

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14165–14178.

Changyoon Lee, Yeon Seonwoo, and Alice Oh. 2022. CS1QA: A dataset for assisting code-based question answering in an introductory programming course. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2026–2040, Seattle, United States. Association for Computational Linguistics.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

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

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Chenxiao Liu and Xiaojun Wan. 2021. CodeQA: A question answering dataset for source code comprehension. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2618–2632, Punta Cana, Dominican Republic. Association for Computational Linguistics.

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. *Preprint*, arXiv:1907.11692.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019,*

New Orleans, LA, USA, May 6-9, 2019. OpenReview.net.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv preprint*, abs/2102.04664.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 1287–1293. AAAI Press.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2010. Exploring the limits of transfer learning with a unified text-to-text transformer. *ArXiv preprint*, abs/10.5555.

Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 784–789, Melbourne, Australia. Association for Computational Linguistics.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas. Association for Computational Linguistics.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

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

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.

Yue Wang, Weishi Wang, Shafiq 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 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. Answerbot: Automated generation of answer summary to developers' technical questions. In *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*, pages 706–716. IEEE.

Yi Yang, Wen-tau Yih, and Christopher Meek. 2015. WikiQA: A challenge dataset for open-domain question answering. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2013–2018, Lisbon, Portugal. Association for Computational Linguistics.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium. Association for Computational Linguistics.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10197–10207.

A Appendix

A.1 Datasets

Dataset	CodeQA		CS1QA
	Java	Python	Python
Train	95,778	56,085	5,542
Valid	12,000	7,000	1,847
Test	12,000	7,000	1,847
Avg. tokens in code	119.52	48.97	83.87
Avg. tokens in question	9.48	8.15	15.73
Avg. tokens in answer	4.74	4.07	19.60
Avg. API Func. in code	6.60	5.06	7.07
Avg. tokens in API Doc.	27.56	21.95	34.61

Table 6: Statistics of CodeQA and CS1QA datasets.

CodeQA Dataset is a dataset for question-answering designed for the comprehension of source code (Liu and Wan, 2021). It consists of two programming languages, Java and Python, containing 119,778 and 70,085 question-answer pairs, respectively.

CS1QA Dataset is a dataset designed for code-based question answering in a real-world classroom setting (Lee et al., 2022). It consists of 9,237 question-answer pairs collected from chat logs in an introductory Python programming class. Each question in the dataset includes the student's code, as well as the position of the code related to the question.

A.2 Evaluation Metrics

- **BLEU** (Lin and Och, 2004) is a BiLingual Evaluation Understudy to evaluate the quality of generated answers. Liu and Wan (Liu and Wan, 2021) used Corpus BLEU (Papineni et al., 2002), but higher order n-grams may not overlap because the generated answers are short. We solve this problem by using smoothed sentence BLEU, which is widely used in PLMs's document generation tasks (Feng et al., 2020; Guo et al., 2020; Ahmad et al., 2021; Wang et al., 2021).

- **METEOR**(Banerjee and Lavie, 2005) is used to measure the correlation between the metric scores and human judgments of translation quality.

- **ROUGE-L**(Lin, 2004) is used to apply the Longest Common Subsequence in code question answering evaluation.

A.3 Analysis of Answer Lengths

In Figure 6, we analyze the performance with respect to answer lengths on CodeQA datasets. Specifically, there are 822 /562 instances with an answer length of 1, 9656 /5887 instances with an answer length between 1-10, 1237 /502 instances with an answer length between 10-20, and 229 /47 instances with an answer length over 20. Our model shows better performance across all answer lengths for both datasets. It demonstrates that our SDR and CQI help generate appropriate answers regardless of length.

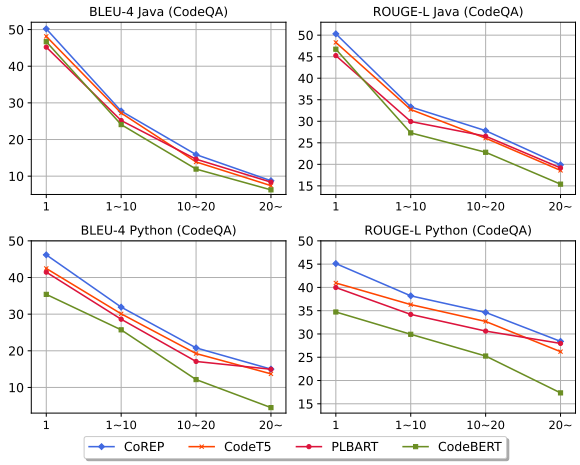

Figure 6: BLEU and ROUGE-L scores with respect to answer lengths (x axis is answer length, y axis is BLEU or ROUGE-L).

A.4 Versatility Across Diverse PLMs

We conducted an experiment to evaluate the applicability of our framework to other PLMs for both code and language. Table 7 presents the results of these experiments, focusing on the performance of the framework. Our model has an end-to-end framework by utilizing selective API document retrieval and question semantic intent clustering. We applied our method to PLBART and CodeGen. We can see that the results improve performance compared to when only code and a question are used as inputs. This demonstrates that our approach can be applied to various PLMs without regard to whether they are encoder-decoder or only decoder.

A.5 LLM based Evaluation

We conduct an LLM-based evaluation on the three datasets to check the quality of generate answers. We adopt the GPT-Rank template for the evaluator (Jiang et al., 2023). We randomly select 1,500 code-question samples and shuffle them. For each

Models	BLEU	METEOR	ROUGE-L
CodeQA (Java)			
PLBART	26.12	12.44	31.07
CodeGen	28.12	13.47	33.38
CodeT5	28.05	13.51	33.57
CodeQA (Python)			
PLBART	29.87	16.34	35.97
CodeGen	31.98	17.27	38.04
CodeT5	32.14	17.72	38.42
CS1QA			
PLBART	7.44	9.86	18.04
CodeGen	6.98	9.35	16.21
CodeT5	7.52	9.77	18.07

Table 7: Versatility across other PLMs for program and language.

	CodeQA (Java)		CodeQA (Python)		CS1QA	
	Flu.	Rel.	Flu.	Rel.	Flu.	Rel.
Win	637	650	625	621	706	761
Tie	279	590	291	280	336	323
Loss	584	260	584	599	458	416

Table 8: LLM evaluation of the appropriateness of the generated answers on three datasets. Flu. and Rel. denote Fluency and Relevance.

selected code-question, we provide two answers generated by CodeT5 and our CoRAC to the LLM evaluator. We use *gpt-3.5-turbo* API for LLM evaluation. When providing the two generated answers, we randomly show them without indicating which model generated them as shown in Table 12. We ask the LLM to evaluate the two following metrics: 1) Fluency (grammatical correctness) and 2) Relevance (selection of the relevant content in source code). The LLM evaluator compares the generated answers with the input code and chooses one of win, tie, and loss in terms of two metrics. Table 8 summarizes the results of LLM-based evaluation on the generated answers on the datasets. The win counts for both fluency and relevance are higher in CoRAC than in CodeT5. In particular, the relevance of CoRAC significantly outperforms that of CodeT5 in all the datasets, indicating that our CoRAC generates more suitable answers.

A.6 Qualitative Analysis

Table 9 and 10 show the answers generated from our proposed model, CoRAC, and baseline models on the CodeQA Java and Python datasets. In Table 10, the answers generated by other baseline models fail to detect the keyword `rendered`. Also, they only include specific words such as html or course info, without fully capturing the overall purpose of the code. On the other hand, our model generates appropriate keywords as answers to the question, which reflects the API function documentation of `render` and `render_to_string`.

Original Code

```
@SuppressWarnings("unchecked")
private Class validateClass(ClientConfig cfg) {
    Class clazz = null;
    try {
        clazz = Class.forName(cfg.getAccessRequestHandlerClassname());
    } catch (final ClassNotFoundException e) {
        LOG.error("Unable to load Handler Class '" + cfg.getAccessRequestHandlerClassname()
                + "' for RADIUS client '" + cfg.getName() + "'. Requests from this client will be ignored.", e);
        return null;
    }
    Object inst = null;
    try {
        inst = InjectorHolder.getInstance(clazz);
    } catch (ConfigurationException | ProvisionException e) {
        LOG.error("Unable to instantiate Handler Class '" + cfg.getAccessRequestHandlerClassname()
                + "' for RADIUS client '" + cfg.getName() + "'. Requests from this client will be ignored.", e);
        return null;
    }
    AccessRequestHandler handler = null;
    try {
        handler = (AccessRequestHandler) inst;
    } catch (final ClassCastException e) {
        LOG.error("Unable to use Handler Class '" + cfg.getAccessRequestHandlerClassname()
                + "' for RADIUS client '" + cfg.getName() + "'. Requests from this client will be ignored.", e);
        return null;
    }
    return clazz;
}
```

Question: What does this return?

CodeT5: access request handler
ChatGPT: an access request handler
CoRAC: the access request handler interface
Gold: the proper interface

Table 9: An example of the qualitative comparison on Java dataset (CodeQA).

Original Code

```
def get_course_info_section(request, user, course, s_key):
    info_module = get_course_info_section_module(request, user, course, s_key)
    html = ''
    if (info_module is not None):
        try:
            html = info_module.render(S_VIEW).content
        except Exception:
            html = render_to_string('error-message.html', None)
            log.exception(u'Error rendering course_id=%s, s_key=%s', unicode(course.id), s_key)
    return html
```

Question: What does this return?

CodeT5: the html
ChatGPT: This function return the rendered html content representing the course information section
CoRAC: the snippet of html to be rendered on the course info section
Gold: the snippet of html to be rendered on the course info page

Table 10: An example of the qualitative comparison on Python dataset (CodeQA).

Original Code

```
from cs1robots import *

load_world("worlds/hurdles1.wld")
hubo = Robot()
hubo.set_trace('blue')
hubo.set_pause(0.3)

def turn_right():
    for i in range(3):
        hubo.turn_left()

def hurdle():
    hubo.turn_left()
    hubo.move()
    turn_right()
    hubo.move()
    turn_right()
    hubo.move()
    hubo.turn_left()

for i in range(4):
    hubo.move()
    hurdle()
    if i == 3:
        hubo.move()
        hubo.pick_beeper()
```

Question: Please explain why you used the if statement in task2

CodeT5: If i is 3, the beeper is picked up if there is a beeper in its place after Hubo moves, and if i is 4, the beeper is picked up if.
ChatGPT: The if statement in task2 is used to check if the robot hubo is on the last hurdle (the fourth one in this case).
CoRAC: If the if statement is used, the robot moves one space, picks it up, and moves it another space.
Gold: To pick up the beeper after crossing the last hurdle.

Table 11: An example of the qualitative comparison on CS1QA dataset.

Template	**Instruction** Please read the original text and the two adversarial texts (Candidate-A and Candidate-B), then evaluate and rank texts generated by two different methods. **Original Text** {*orig_text*} **Generated Answers** **Candidate-A** : {*generated_text1*} **Candidate-B** : {*generated_text2*} **Questions** Given the instruction and input above, please compare the two candidates based on the {*metric*}. "{*metric*}" {*metric_desc*} You only have 3 choices to output: If you think A is better, please output: 1. Candidate-A is better If you think B is better, please output: 2. Candidate-B is better If you think A and B are tie, please output: 3. tie Do not output anything else except the 3 choices above. **Output your choice below Comparison Option (1 or 2 or 3)** 1. Candidate-A is better 2. Candidate-B is better 3. tie
Variables	{*orig_text*} is original sentence. {*generated_text*} is adversarial generated examples by attack methods. {*metric*} is metric to evaluate the quality of generated answer. we use two metrics as follows: `Fluency` and `Relevance`. {*metric_desc*} is description of the metric. The description is paired with the following two metrics: `Fluency` : "evaluates the generated answer is grammatical correctness." `Relevance` : "evaluates the relevance between the generated answer and the given code and question."

Table 12: Prompt for LLM evaluation. We adopt GPT-rank template for the evaluation prompt. To ensure a fair comparison, we randomly select 1,500 examples that were generated by CodeT5 and CoRAC. We use `gpt-3.5-turbo` API. To aovid naming bias, we use "candidate-A" instead of the method's name to avoid naming bias. Additionally, the generated answer were randomly assigned to the {*generated_text1*} and {*generated_text2*} positions to prevent positional bias.